分散メモリ型並列計算機のための交互方向操作に基づく処理系とアルゴリズムの研究

今村俊幸

分散メモリ型並列計算機のための交互方向操作に基づく処理系とアルゴリズムの研究

1999年3月

今村 俊幸

論文概要

本論文は分散メモリ型並列計算機のための交互方向操作を基本とする並列計算モデル ADEPS を首尾よく表現するための並列処理言語 ADETRAN4 処理系を広範な計算機システム上に実装したことの内容を論述し, 科学技術計算の重要な項目である線形方程式解法における LU 分解, 固有値計算におけるハウスホルダー変換法並びに divide and conquer 探索法の並列計算法を提案し評価を行っている. また応用問題に対する適用に関して並列化並びにその試験結果および評価を詳しく述べている.

科学技術の発展により並列処理を行うハードウェアは我々の身近に存在しているが, ソフトウェア環境は身近なものとは言い切れない. その背景には並列計算に必要な, 1) 問題, 2) アルゴリズム, 3) 並列処理モデルの 3 項目を統一的に取り扱う必要があるにもかかわらず, その枠組みが確立していないことが問題であると考える.

本論文で取り上げた並列処理言語 ADETRAN4 は交互方向操作に基づく並列処理モデル ADEPS を持っており多くの問題を表現する能力に優れている. 本研究では特に分散メモリ型並列計算機上での処理系として ADETRAN4 の実装とともに, 並列アルゴリズムの表現に必要と思われる拡張を施している.

第 2 章においては, 本研究で扱う並列アルゴリズムの表現ならびに実験において利用する並列計算言語 ADETRAN4 の開発経緯および概要を示している. また, 並列計算言語 ADETRAN4 の並列計算機 ADENART, VPP-500 上での実装を述べるとともに, MPI と Fortran90 を用いたプリプロセッサを実装することにより殆んどの並列計算機上で ADETRAN4 が利用可能になったことを示している. 同時に, 分散メモリ上での並列アルゴリズム記述のために必要な構文拡張を行っている.

第 3 章では, 行列計算の典型的な例として挙げられる線形方程式求解のための LU 分解, 固有値計算で利用されるハウスホルダー 3 重対角化, 固有値探索のための divide and conquer 法を取り上げ, 並列化並びにコストモデルの導出を通じてその並列処理の効果を議論している. また, それら問題で新たなアルゴリズムを提案し, コストモデルや並列計算機上の実験から優位性を示している.

第 4 章では, NAS パラレルベンチマークと粒子プラズマコードの 2 つを取り上げ 大規模応用問題に対する並列化の議論を行っている. NAS パラレルベンチマークでは, 並列計算機の基本的な性能を計る 5 つのカーネルコードと典型的な 3 つの流体コードに対して ADEPS による並列化の結果を示している. 粒子プラズマコードでは, 粒子コード特有の計算部分の並列化の困難さを示すとともに粒子分割, 領域分割に基づく並列化手法を示した. また, 5 種類の並列計算機上での数値実験を通してプログラム並びに計算機の特性に関する議論を行っている.

目次

第 1 章

序

1.1 並列計算機の現状と問題点

半導体技術の進歩により計算機の演算能力が飛躍的に向上している．西暦 2000 年までには単体性能で 1GFLOPS(1 秒間に 10 億 (=10^9) 回の浮動小数点演算を行う能力) を越える汎用の MPU(マイクロプロセッサユニット) が登場し実用化されると云われている．一方, 科学技術計算の最先端では TFLOPS [1] 超級の演算能力が必要とされ, それを実現するための手段として高性能プロセッサを並列に動作させる並列計算機の利用が必須のものとなっている．

並列計算機の開発の歴史は古く, 様々な実装の試みがみられる．古くは, 恐らく現在の並列処理の始まりといわれている “リチャードソンの夢”(1922 年) が有名なところであろう．しかしながら, アーキテクチャや言語仕様等が特殊であったり, 専用用途に限定されたものとして開発されたために広く一般に普及するものがなかった．現在, ハードウェア技術的にも高性能な並列計算機を構成することは困難ではなく, TFLOPS を越える並列計算機の開発プロジェクトとして本国科学技術庁における “地球シミュレータ計画” ならびに米国 DOE の “ASCI(Accelarated Strategic Computing Initiative) 計画” などが進められている．スーパコンピュータがかつてベクトル計算機中心であったものから並列計算機にとって替わられたのは, 主要な研究機関の計算機資源を報告する “TOP500 レポート”[13] [2] の内容からも明らかである．また, Pentium の様なパーソナルユースの計算機で用いられている MPU が中規模程度のサーバマシンに使用され, マルチプロセッサモデルが当然の様に採用されている．つまり, 並列計算機のハードウェアそのものは広範囲に普及しているとはいえないが, 基幹処理には必須のものとなっていると言って良い．

一方, ソフトウェアの側面から見た場合の並列計算機の成熟度は十分であろうか．利用者が新たに並列計算機を利用し並列プログラム開発を行う場合, それを行うための言語およびライブラリ等を含む周辺ソフトウェアが重要である．また, 並列計算機が普及するための重要な項目として考えなくてはならないはずである．近年, 並列プログラム開発の最も有効な手段として MPI, HPF, OpenMP の 3 つの標準化がなされている．それぞれの特徴と現状は次の様に簡単にまとめられる．

[1] 1TFLOPS=1 秒間に 10^{12} 回の浮動小数点演算を行う能力, 更にその上の単位として PFLOPS(P=10^{15}) がある．

[2] 引用文献は 1997 年のものだが, `http://www.top500.org` に最新情報が定期的に報告されている．

- MPI[24, 25] はメッセージパッシング手法を行うために使用する通信関数の仕様であり, 広く認知されている. メッセージパッシング手法は, プロセス単位に独立したメモリ空間を管理しながらデータの必要の有無に応じて`send`と`recv`の対によってデータの送受を行うものである. MPI は集合通信など高度な通信方法をサポートしており, きめ細かな並列処理が可能である. 一方, 配列の分割方法や分割後のデータ管理, 通信デッドロック回避などをプログラマの責任の下で行わなくてはならない. この様な意味で, メッセージパッシング手法は並列処理のアセンブラともいわれており, 並列プログラム作成には高度なスキルが要求される.

- HPF[17, 18] はデータ並列性を表現するための言語である. データ並列は, メッセージパッシング手法と異なり配列データをグローバルな視点から扱うため, 並列プログラムの字面上の様相が従来プログラムと差異が少ない. また, 配列の分割やデータ管理を言語処理系が行うため, プログラマがそのための処理をプログラム中に書く必要がない. そういった意味で, HPF の様なデータ並列を記述するための並列計算言語が普及することが期待されていた. しかしながら, 処理系の実装が遅れたためと言語そのものが表現し得るアルゴリズムの範囲が狭かったため, 一般に認知されるには至っていない. 未だ発展途上の状況と言える. [59].

- OpenMP[35] はこれまで言語仕様の標準化がなされて来なかった共有メモリ型並列計算機の標準言語として広く認知されつつある. OpenMP はコンパイラに補助的指示を行うためのプラグマもしくはディレクティブ (指示子) と幾つかの関数, 環境変数を定義したものである. 最新の並列計算機のハードウェアの構成では, 複数のプロセッサがメモリ共有するノードを構成し複数のノードを高速ネットワークで結合する分散共有アーキテクチャが多くなってきており OpenMP の果たす役割は大きいといえる.

かつてのスーパコンピュータであるベクトル計算機が登場したときには, ベクトル計算機向けにコンパイラが最適化を行う自動ベクトル化コンパイラ (もしくはベクトライザ) が存在していた. 並列計算機上の自動並列化コンパイラは, 実用レベルに達しているものはなく HPF, OpenMP などの指示子が必要である. 以上の経緯から判断されることは、並列処理のハードウェアは我々の身近に存在しているが, それを利用者が意図して使うための並列計算言語, 自動並列化コンパイラ等のソフトウェア環境は十分なものとはいえないであろう.

この様な背景には, 並列計算を行うまでに取り扱わなくてはならない “問題 - アルゴリズム - 並列処理モデル” の 3 項目を統一的に扱う枠組みがないことに問題がある. 京都大学野木が中心となって開発した ADETRAN4[30, 33] は, HPF の様な FORTRAN の拡張型言語であるが, HPF とは異なり, 明確な並列処理モデル ADEPS (Alternating Direction Execution Parallel over Segment) を持っている. その並列処理モデルは, ADI を代表とした様々な問題を解く上での計算スキームから発展したもので, 問題表現能力に優れている [60, 61]. HPF は owner computes rule と, 分散するデータの CopyIn-CopyOut タイミングを規定したモデル (以下 HPF モデルと

いう) を持っているが, プログラムを記述する立場ではなく, むしろプログラムを処理する立場のモデル設定がされていた. HPF の原型となった, Fortran-D[23] や Vienna-Fortran[42] も同様である. また, その様な立場をとったため HPF は HPF モデルの上に様々な並列処理モデルを表現しようと様々な拡張を加え非常に複雑な仕様となっている. 複雑な仕様, HPF モデルの実装の困難さが HPF 利用ならびに普及の妨げになったと考えられる. 利用者の立場からは, 明快かつ単純な言語仕様であることが重要と考える. その次に, 拡張が容易に行えるような枠組みが用意されていることが理想である. ADETRAN4 は明確な並列処理モデルを前提としているため, 言語仕様も単純であり, 処理系を様々な並列計算機上で実現できる.

並列処理を行う上で考察しなくてはいけないもう一つの観点として並列アルゴリズムがある. 従来の逐次計算機上で利用されてきたアルゴリズムはそれが自然に内包する並列性を利用して並列処理がなされるわけであるが, アルゴリズムを逐次的な表現にしたために並列性が失われたり, 人工的に構成されたアルゴリズムの中にはデータの依存関係から並列性を持たないものが存在する. その様なアルゴリズムは, 並列処理を行うために並列性の高い別のアルゴリズムに置き換えなくてはならない. そして, そのアルゴリズムをプログラムとして表現する際に, アルゴリズムの持つ並列性を損なわないで表現可能な並列言語が望まれる. そういった意味で, 並列言語には HPF や ADETRAN4, OpenMP の様に並列性を指示できる拡張言語であることが要求される. また, アルゴリズムの計算モデルを導出しその性質を実際に検証することが工学的に重要な意味を持つが, 計算モデルの中に含まれるアルゴリズムの性質を表現する幾つかのパラメータを, プログラミング言語で自由に表現可能かという問題がある. さらに, それらパラメータの制御が容易に行えるかも重要な問題である. これらを満足し, しかも並列アルゴリズムの特性を研究議論する上で各種多様な並列計算機で処理可能な言語が重要である.

現在, これらを満足する言語系の内 HPF,OpenMP は処理系の実装が広範囲に及んでいないということで, メッセージパッシングモデルが選択肢として残るわけであるが, 先に述べた通りプログラム開発者に高いスキルを要求する等の問題がある. ADETRAN4 は比較的単純な処理モデルのもと, 簡単な言語仕様で並列プログラムを開発できる利点がある. また本論文で示すように, 様々な並列計算機で実装され利用することが可能である. そして, 問題 – アルゴリズム – 並列処理を一括して捉えることができる. 本論文ではこの利点に着目し, 処理系ならびにアルゴリズムの側面で研究を行った.

1.2 本論文の構成

本研究は, 特に分散メモリ型並列計算機上での処理系として, 並列計算言語 ADETRAN4 の実装を行うとともに, 並列アルゴリズムの表現に必要と思われる拡張を施した. また, 典型的な行列計算の並列化を開発した処理系を用いて行い, 並列アルゴリズムのモデル化, 並列化効率の議論, 新アルゴリズムの提案を行っている. さらに, 実用規模の応用問題に対して考察も行っている.

次章からの内容をここで簡単に示す. 第 2 章においては, 本研究で扱う並列アルゴリズムの表現ならびに実験において利用する並列計算言語 ADETRAN4 の開発経緯および概要を示している.

また, 並列計算言語 ADETRAN4 の並列計算機 ADENART, VPP-500 上での実装を述べるとともに, MPI と Fortran90 を用いたプリプロセッサを実装することにより殆んどの並列計算機上で ADETRAN4 が利用可能になったことを示している. 同時に, 分散メモリ上での並列アルゴリズム記述のために必要な構文拡張を行っている.

第 3 章では, 行列計算の典型的な例として挙げられる線形方程式求解のための LU 分解, 固有値計算で利用されるハウスホルダー 3 重対角化, 固有値探索のための divide and conquer 法を取り上げ, 並列化並びにコストモデルの導出を通じてその並列処理の効果を議論している. また, それら問題で新たなアルゴリズムを提案し, コストモデルや並列計算機上の実験から優位性を示している.

第 4 章では, NAS パラレルベンチマークと粒子プラズマコードの 2 つを取り上げ 大規模応用問題に対する並列化の議論を行っている. NAS パラレルベンチマークでは, 並列計算機の基本的な性能を計る 5 つのカーネルコードと典型的な 3 つの流体コードに対する ADEPS と呼ばれる処理様式での並列化について示している. 粒子プラズマコードでは, 粒子コード特有の計算部分に関する並列化の困難さを示すとともに粒子分割, 領域分割に基づく並列化手法を示した. また, 5 種類の並列計算機上での数値実験を通してプログラム並びに計算機の特性に関する議論を行っている.

第 5 章では, 本論文の総括を行っている.

第 2 章

ADETRAN 4処理系の実装

本章では並列プログラム開発のためのプログラミング環境整備として並列プログラミング言語 ADETRAN4 の処理系開発について述べる．ADETRAN4 は独自の並列処理様式 ADEPS に基づいた並列モデルを記述することに適した言語して開発された．ADETRAN4 は ADEPS モデルを表現することに特に適しているが，著者の研究によりそれ以外のモデルであるメッセージパッシングモデルをも記述できるような処理系の拡張が施され，様々なプログラムの記述ならびに，記述したプログラムが様々な並列計算機上で実行可能となっている．

2.1 ADETRAN4 言語の概要

並列計算言語 ADETAN4 の前身である ADETRAN[30] は，京都大学と松下電器産業の協力によって開発された分散メモリ型並列計算機 ADENART 用のプログラミング言語である．並列計算機 ADENART は，京都大学野木教授によって考案された並列計算機アーキテクチャ ADENA[29] の試作機であり，計算機アーキテクチャ ADENA は独特のネットワーク結合により，分散配置された配列データの軸回転的な再分散を高速に行うことができるという特徴を持っていた [22]．また，高速な軸回転が可能であるというハードウェア的な特徴により，後節において説明する並列処理様式 ADEPS を効率的に実行できるアーキテクチャであった．

並列計算機 ADENART 上で利用される並列計算言語 ADETRAN は科学技術分野で使われる FORTRAN 言語に近い文法を持ち，ADENART で実現可能な高速なデータ再分散を記述するためのデータ構造と構文を定義していた [33]．若谷 [63] らによって ADETRAN 処理系が開発され，ADETRAN プログラムの開発環境ならびに FORTRAN から ADETRAN へのトランスレータ等も開発された [53]．更に，様々なアプリケーションが移植され成果をあげたという報告がある [34, 51, 41]．

ADENA の後継モデルとして，分散メモリに代わり共有メモリバンクを使ってハードウェアを構成した ADENA4 が考案されている [32]．ADENA4 は ADENA で行っていたネットワークを介したデータ再分散を行う代わりに，共有メモリバンクからインタリーブにアクセスする経路を複数用意しその切り替えによって配列の再分散に対応する動作を実現するというものである．ADENA4 上での利用を目的とし，前身の ADETRAN からデータ再分散等の構文を取り除き幾つかの

拡張を加えたものが本論文で取り扱う並列計算言語 ADETRAN4 である．本節以下, ADETRAN が対象とする並列処理モデル ADEPS を述べ, ADETRAN4 の言語の文法等について述べる．

2.1.1 並列処理モデル : ADEPS

ADENA4 の並列処理は **ADEPS** (**A**lternating **D**irection **E**xecution of '**P**arallel over **S**egments') と呼ばれる並列処理モデル [60] に基づく．ADEPS は物理現象を表現する 2 次元又は 3 次元配列をある方向から見た 1 次元部分配列 (セグメント) の集合とみなし, それを処理の基本単位とし計算するものである．

まず, 行列計算を取り上げ説明を行う．行列の処理は, 様々な要素を用いることによって実現されるが, 最終的には行方向から行列を見て列ベクトルの集合と見なし複数の列ベクトルに関する処理を並列に処理するもの見なすことができ, 列方向から行列を見て行ベクトルの集合と見なし複数の行ベクトルに関する処理を並列に処理するものと見なすことができる．この様に ADEPS では, 行列の処理を行と列の 2 つの処理形態に分解し, 2 つの状態が交互に現れた場合には, '行' と '列' の 2 方向の処理をダイナミックに切替えながら処理を進行していく．

次に, 2 次元の拡散方程式を有限差分法により時間積分する場合を取り上げる．陽的スキーム

$$U_{i,j}^{k+1} = U_{i,j}^{k} + \frac{\Delta t}{(\Delta x)^2}\left(\delta_x^2 U_{i,j}^{k} + \delta_y^2 U_{i,j}^{k}\right) \tag{2.1.1}$$

$$\delta_x^2 U_{i,j} \equiv U_{i-1,j} - 2U_{i,j} + U_{i+1,j} \tag{2.1.2}$$

$$\delta_y^2 U_{i,j} \equiv U_{i,j-1} - 2U_{i,j} + U_{i,j+1} \tag{2.1.3}$$

では,k に関する 1 ステップが 2 段階に分解可能であり, '$i-1$', '$i+1$' の添字部分を x 方向の sweep, '$j-1$', '$j+1$' の添字部分を y 方向の sweep として

$$U_{i,j}^{k+\frac{1}{2}} = \delta_x^2 U_{i,j}^{k} \tag{2.1.4}$$

$$U_{i.j}^{k+1} = U_{i,j}^{k} + \frac{\Delta t}{(\Delta x)^2}\left(\delta_y^2 U_{i,j}^{k} + U_{i,j}^{k+\frac{1}{2}}\right) \tag{2.1.5}$$

と書き換えることができる．この 2 式は第一式 (2.1.4) が x 方向のセグメント操作, 第二式 (2.1.5) が y 方向のセグメント操作である．

陰的スキームとしての ADI 法

$$U_{i,j}^{k+\frac{1}{2}} = U_{i,j}^{k} + \frac{\Delta t}{2(\Delta x)^2}\left(\delta_x^2 U_{i,j}^{k+\frac{1}{2}} + \delta_y^2 U_{i,j}^{k}\right) \tag{2.1.6}$$

$$U_{i,j}^{k+1} = U_{i,j}^{k+\frac{1}{2}} + \frac{\Delta t}{2(\Delta x)^2}\left(\delta_x^2 U_{i,j}^{k+\frac{1}{2}} + \delta_y^2 U_{i,j}^{k+1}\right) \tag{2.1.7}$$

では, 第一式, 第二式ともに陽解法の場合と同様な書き換えを行い, 下式のようにまとめれば, x 方向, y 方向のみのセグメント操作に帰着することができる．

$$U_{i,j}^{k+\frac{1}{4}} = \delta_y^2 U_{i,j}^{k} \tag{2.1.8}$$

$$U_{i,j}^{k+\frac{1}{2}} = U_{i,j}^{k} + \frac{\Delta t}{(\Delta x)^2}\left(\delta_x^2 U_{i,j}^{k+\frac{1}{2}} + U_{i,j}^{k+\frac{1}{4}}\right) \tag{2.1.9}$$

$$U_{i,j}^{k+\frac{3}{4}} = \delta_x^2 U_{i,j}^{k+\frac{1}{2}} \tag{2.1.10}$$

$$U_{i,j}^{k+1} = U_{i,j}^{k+\frac{1}{2}} + \frac{\Delta t}{2(\Delta x)^2}\left(U_{i,j}^{k+\frac{3}{4}} + \delta_y^2 U_{i,j}^{k+1}\right) \tag{2.1.11}$$

3次元問題についても同様にして, x方向, y方向, z方向のセグメント単位の処理に帰着することができる. 3次元の例として3次元データの離散フーリエ変換を取り上げる.

$$\hat{U}_{k_x,k_y,k_z} = \frac{L_x}{N_x}\frac{L_y}{N_y}\frac{L_z}{N_z}\sum_{j_z=0}^{N_z-1}\sum_{j_y=0}^{N_y-1}\sum_{j_x=0}^{N_x-1} u(x_{j_x}, y_{j_y}, z_{j_z}) W_x^{k_x} W_y^{k_y} W_z^{k_z} \tag{2.1.12}$$

ここで

$$W_A = \exp(-2\pi i/N_A), \quad A_{j_A} = (L_A/N_A) j_A \quad (\text{ただし} A \text{は} x, y, z)$$

x,y,zの各方向毎には次の様に分解できる.

$$u_{k_x}(y_{j_y}, z_{j_z}) = \frac{L_x}{N_x}\sum_{j_x=0}^{N_x-1} u(x_{j_x}, y_{j_y}, z_{j_z}) W_x^{k_x} \tag{2.1.13}$$

$$u_{k_x,k_y}(z_{j_z}) = \frac{L_y}{N_y}\sum_{j_y=0}^{N_y-1} u_{k_x}(y_{j_y}, z_{j_z}) W_y^{k_y} \tag{2.1.14}$$

$$\hat{U}_{k_x,k_y,k_z} = \frac{L_z}{N_z}\sum_{j_z=0}^{N_z-1} u_{k_x,k_y}(z_{j_z}) W_z^{k_z} \tag{2.1.15}$$

この様に, 並列処理モデルADEPSは計算過程に現れる様々な計算スキームを(例えばユークリッド空間における)自然な形の直交座標方向に関する1次元化と作用素分解によってアルゴリズムを構成しようとするモデルである.

このアルゴリズムの構成を行う際に, ADEPSには作用素分解対象として取り扱うための大域的なデータのスコープと, 1次元化によって生じる局所的なデータのスコープが存在する. このスコープが, 並列計算機上での実現においてはグローバルな変数とローカルな変数またはセグメント化された配列とベクトルというADETRAN4特有のデータ構造として現れてくる. 詳しくは, 次節で述べるADETRAN4構文で解説する.

2.1.2 ADETRAN 4構文

次にADETRAN 4について説明する. ADETRAN 4は並列処理様式ADEPSを効率的に表現するために開発された言語であり, 以下に述べていく特徴を持つ. ADETRAN4全構文規則に関してては付録Aに示している.

基本データ構造

ADETRAN 4の持つデータ構造は2もしくは3次元のセグメント配列とセグメントベクトルの2つに大別される.

セグメント配列　前者のセグメント配列は, 全データを分散して持ち, システム全体で一つの配列と見るものである. 表記上はFORTRANでの表記方法と同じであるが, 担当プロセッサを明示するスラッシュ対をオプションで記述することができる. システム全体で意味ある配列を形成するグローバルなスコープを持つと同時に, インデックスに付加されたスラッシュ対で1次元化されたローカルな配列としてのスコープを併せ持つデータ構造と云える.

3次元配列aを例に挙げれば, FORTRANで次の様に表記したものを

```
a(i,j,k)
```

ADETRAN4では

```
a(i,j,k) または a(i,/j,k/)
```

の様に表す. オプションであるスラッシュ対の指定方法は3通りある. a(i,/j,k/)はx方向配列と呼ばれ, 論理プロセッサ /j,k/ が所有する1次元配列a()のi番要素を表す. a(i/,j,/k)はy方向配列と呼ばれ, 論理プロセッサ /k,i/ が所有する1次元配列a()のj番要素を表す. a(/i,j/,k)はz方向配列と呼ばれ, 論理プロセッサ /i,j/ が所有する1次元配列a()のk番要素を表す.

ここで論理プロセッサ /j,k/ は, 第一インデックス'j'がベクトルプロセッサによって処理される論理番号を表し, 第二インデックス'k'がプロセッサの論理番号を表す. 実装システムのアーキテクチャに応じて, ('j', 'k')の組でもってプロセッサの論理番号を表すこともある. ベクトルプロセッサでない場合には, セグメント配列は各プロセッサに1次元配列の集合, つまり2次元配列として割り当てられる.

また, 3次元以上のデータ構造を2または3次元配列の集合と考えて扱うための拡張次元を定義している. 前述の3次元配列a(i,j,k)に後続して括弧対を付加することで, 高次元配列を扱うことができる. この括弧対のことを主となる3次元部分 (主次元) に対して**拡張次元**と呼ぶ. FORTRANでの6次元配列

```
a(i,j,k,l,m,n)
```

は, ADETRAN4では次の様に表記される.

```
a(i,j,k)(l)(m)(n)
```

セグメントベクトル　もう一つのデータ構造であるセグメントベクトルはプロセッサ間で共通なデータセットとして定義される. ただし, 2次元または3次元配列の表記をセグメント配列と区別するために全てのインデックスを括弧対にして列挙することとしている. 例えば, 3次元のセグメントベクトルは

```
a(i)(j)(k)
```

の様に表記される. 本表記手法はC言語での表記方法と酷似しているが, インデックスの増分によるアドレスの増分はインデックスの右側からではなく, 左側から1つづつ増加するFORTRANで

のメモリ割り当て方式と同様とする．また, 配列ではない単純変数についてもセグメントベクトルとして分類する．

また, セグメント配列は論理プロセッサ /j,k/ に対して 1 つの変数領域が割り当てられる **poly** 変数ではなく, 共通メモリセット上に 1 つの変数領域が割り当てられる **moly 変数** [41] である．セグメントベクトルは各論理プロセッサから定義・参照可能であるため, 同一物理プロセッサに割り当てられた論理プロセッサ間でのデータ送受信に利用することができる．

さらに, セグメントベクトルはある実行文を処理する時点で物理プロセッサ全体で等しい値を保持するか否かで**グローバル変数**または**ローカル変数** に分類される．ADETRAN4 は制御構文のカウンタ等に単純変数を記述するため, どの物理または論理プロセッサが所有する値を利用するかが問題となるが, 制御構文に利用されるカウンタ類にはグローバル変数から構成されるグローバルな変数値に限定されることをここで注意しておく．ローカル変数は後述する PDO ブロック内での並列計算の一時変数に使われる．

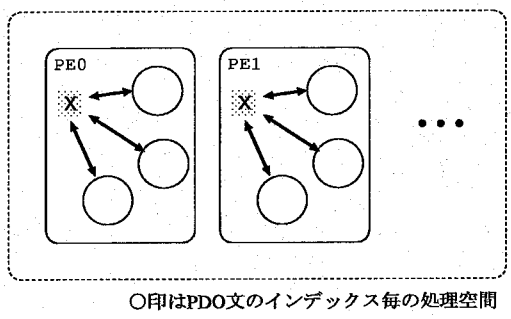


図 2.1: moly 変数

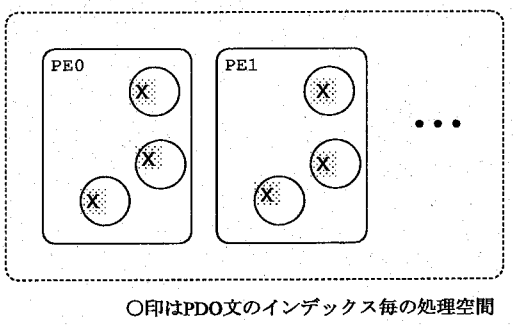


図 2.2: poly 変数

並列処理方式 ADEPS では方向毎のデータアクセスが中心となる．方向毎にデータを格納する方式は ADETRAN で採用されていたが, 異なる方向で定義された配列要素を参照するためには PASS 文と呼ばれる構文を用いて転送 (再分散) する必要があった．ADETRAN4 でも, 実装するシステムよってデータの再分散が必要であるが, 分割されたデータは軸方向毎のアクセスに対しローカルとなるようコンパイラ側でデータ転送を行なうなど整合性が保たれるため, ユーザーは軸方向毎に配列を確保する必要も, 軸方向を変えることで生じる再分散 (re-distribution) やデータ転送等に必要な構文の記述も必要としない．

構文階層とサブプログラム階層

ADETRAN 4 はホスト – スレーブ方式におけるスレーブ部分のプログラムを記述するものである．ADETRAN 4 の構文は, システム全体のグローバル制御 (G), 並列実行制御指示 (P), 各プロセッサ単位のローカルな制御ならびに実行文 (L) を行う 3 階層の構文に分類されており, G-P-L が各構文の先頭文字につき区別される．

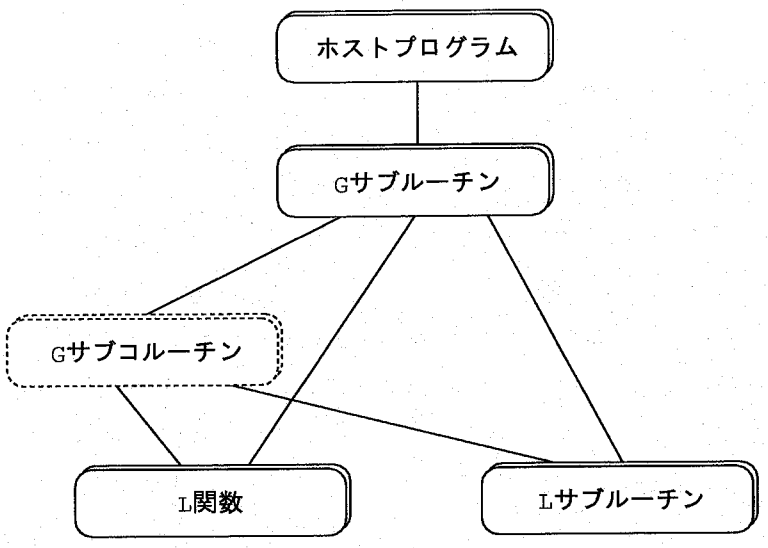


図 2.3: サブプログラムの階層構造

[G] GDO, GIF, GGOTO, GCALL, IFALL/IFANY 文

[P] PDO, PCAST 文

[L] FORTRAN77 における実行文 (入出力を除く)

これら階層化された構文はそれ以下の階層を含むことで全体が形成される．またこれら階層に対応したサブプログラムがあり, 図 2.3の関係をなす.

PDO~PEND ブロック

ADETRAN4 では, 並列計算の実行は **PDO~PEND** ブロック内で行なわれる．つまり並列処理の基本は do ループの並列処理を行うデータ並列である．個々の PDO ブロック内は一つの方向属性を持ち, その方向と同じ属性を持つ配列のみ計算がなされる．図 2.4は典型的な 2 次元拡散スキームを ADETRAN 4 で記述したものである．このプログラムの様にユーザーは同一方向属性にあるセグメントのインデックスを自由に変化させながらプログラミングを行う．そして残りのインデックスは並列の分割かまたはベクトルの実行ループとされる．また, HPF などと異なり, スカラ変数はループインスタンスに関わらずループ外でも有効である．さらに, PDO ブロックの前後において積極的な同期を保証しない.

GDO, GIF, GGOTO 文

GDO, GIF, GGOTO 文はともに全ての物理プロセッサで同じ動作を行うことを保証した制御構文である．ただし, これら構文の前後でプロセッサ間の同期がとられるわけではなく, 全てのプロセッサの基本ブロック単位での実行順序を一致させるためのグローバルな構文に過ぎない．文面上の実行順序は一致しても, 実行している時刻はプロセッサによって異なる.

```
pdo  j=1,n
  do  i=2,n-1
    v(i,/j/)=u(i-1,/j/)-2*u(i,/j/)+u(i+1,/j/)
  enddo
pend
pdo  i=1,n
  do  j=2,n-1
    v(/i/,j)=u(/i/,j)+a*((u(/i/,j-1)\
              -2*u(/i/,j)+u(/i/,j+1)+v(/i/,j))
  enddo
pend
```

図 2.4: 2 次元拡散スキーム

1. GDO 文は, 全プロセッサで基本ブロック単位のグローバルな反復を行うことを制御する構文である.
2. GIF 文は, 全プロセッサで指定された論理式の値に応じてグローバルな分岐を行うことを指示する構文である.
3. GGOTO 文は, 全プロセッサでグローバルに制御を移すことを指示する構文である.

図 2.5, 2.6 にこれら構文の例を示す.

```
gdo  i = 1, N
   pdo  j = 1, M
      a(i,/j/) = ....
   pend
   ....
genddo
```

図 2.5: GDO 文の例, GDO~GENDDO の間を全てのプロセッサが N 回反復する

```
      gif ( i != 0 ) then .
         pdo  j = 1, M
            b(i,/j/) = ....
         pend
         ....
      gendif
      gif ( i == 1 ) ggoto 1000
      ....
1000  continue
```

図 2.6: GIF 文, GGOTO 文の例, 全てのプロセッサ間で同一の条件分岐を行い真であれば GIF~GENDIF ブロック内を全てのプロセッサが実行する.

PCAST 文

文献 [31, 60, 61] では様々な問題レベルでの ADEPS による実現を取り上げているが, 交互方向に変数を見ながら DO ループを回すだけでは, 配列の特定場所をアクセスするのに不自由な場合が起きる. それを補うためにデータ編集操作構文である **PCAST 文**が提供されている. PCAST 文は分割されたデータの一部をローカルな配列 (セグメントベクトル) に再分散 (redistribute) しかつ編集 (realign) し直すものである. 例えば行列の転置やリストベクトルを用いたアクセスにおいて, 異なるプロセッサの持つメモリ領域へ参照するものに対して使われる.

PCAST 文は右辺に現われるセグメント配列の方向属性 (コンパイラが行うデータフロー解析によって解決される直前に定義された方向属性) と左辺に現われるセグメントベクトル, および構文に現われる制御カウンタの位置によってその意味合いが異なる. 後述する実装部分でも述べるが, ブロードキャスト, gather-All の転送形式とデータ順序を変更する機能を有している. PCAST 文の右辺が 2 次元のセグメント配列の場合で例を挙げよう.

図 2.7では便宜上, 右辺のセグメント配列a の直前定義方向をスラッシュ対 ('//') によって明示している. この場合, PCAST ブロック内の第 1 文は論理プロセッサ 'j' が所有するセグメント 1:n をその他の物理プロセッサに放送すること意味する. 第 2 文は 1 から n 番目の論理プロセッサの保有するセグメント内の第 i 要素を全ての物理プロセッサに放送し, それを保有元の論理プロセッサ番号順に整列させることを行う. 第 3 文目は左辺のセグメントベクトルに格納する際に, 第 2 インデックスの位置にカウンタが現われるような整列を行う.

```
pcast  i=1,n
  v1(i) = a(i,/j/)
  v2(i) = a(j,/i/)
  v3(k)(i) = a(j,/i/)
pend
```

図 2.7: PCAST 文 (例 1)

```
pcast  i=1,n, j=1,n
  v1(i)(j) = a(i,/j/)
  v2(j)(i) = a(i,/j/)
pend
```

図 2.8: PCAST 文 (例 2)

さらに, 図 2.8では PCAST 文のカウンタを 2 つにした場合を取り上げた. この例は 1 から n 番目の論理プロセッサのセグメントデータを全ての物理プロセッサに放送すること行う. 分散メモリ型計算機では一般的にマルチキャストと呼ばれるデータ転送と対応づけることができる. なお PCAST ブロック内の第 1 文と第 2 文とでは左辺のカウンタの順序が異なっており, PCAST 文以降での変数の参照方法に応じて両者を選択できる形となっている.

2.2 ADETRAN4 処理系実装に関して：準備

本節では ADETRAN4 処理系の実装について述べる．本論文で開発した ADETRAN4 処理系は，開発過程の歴史的な経緯から，対象とする並列計算機システムと変換プログラム言語 (通信ライブラリ) に応じて三世代の処理系実装がある．それらは，次の構成をとり図 2.9にあるような系図をとる．

- 第一世代は，ADENART を対象システムとして入力ソース ADETRAN4→ 出力言語 ADETRAN の変換するadeconv システムである．ADETRAN から ADETRAN4 にアップデートされた際に、追加された構文のエミュレートを実現している．
- 第二世代は，VPP500 を対象システムとして入力 ADETRAN4→ 出力言語 VPP-FORTRAN に変換するシステムである．VPP-FORTRAN は分散メモリ型並列ベクトル計算機 VPP500 での並列プログラムを記述するために開発されたものであり，VPP-FORTRAN が想定する並列モデルへ適切にマッピングし ADETRAN4 固有の構文・処理のエミュレートを実現している．
- 第三世代は，通信ライブラリ MPI(Message Passing Interface) を使用し、ベース言語として FORTRAN90 をサポートするシステムを対象としている。入力 ADETRAN4→ 出力言語 FORTRAN90 + MPI に変換するシステムである．また，ADETRAN4 に対象システムのメモリ分散性を利用者に提供する構文 (POLY 変数の実現，第 2.5.3 節にて示す UNIFY 文，ON 節など) を追加拡張している．

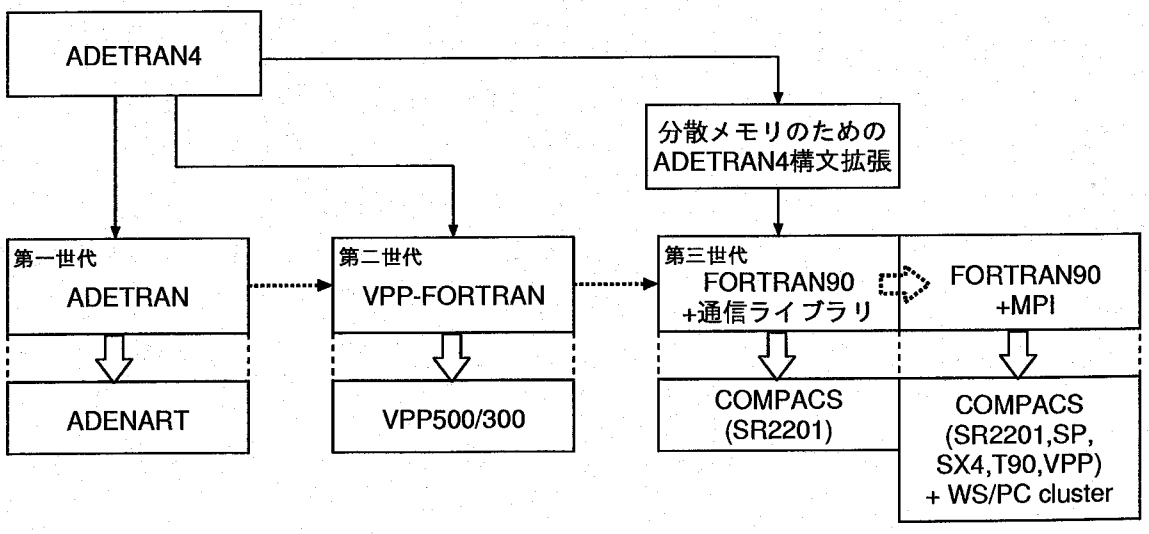


図 2.9: ADETRAN4 処理系系図

処理系実装において、 ADETRAN4 が想定する並列アーキテクチャモデルと、本論文でターゲットとしている並列計算機アーキテクチャモデルの違いをいかにエミュレートするかが最大のポイントとなる．両者の間での差異はメモリモデルにおいて共有分散であるか分散であるかということが最も大きいことである．この差異はコンパイラが管理する変数空間の中で, 交互方向の配列データつまり分割方法の異なる配列データを同一視して管理し吸収する必要がある．そのために, コンパイラは交互方向毎に異なるデータ領域を確保し, または再分散処理を行い, プログラム全体での配列データの整合性をとる必要がある．この処理は, 第一世代のみならず第二, 第三世代においても必要となる処理であり, 第一世代での解析手法が継承されている．

第一世代以下, 第二, 第三世代での一連の処理を説明するために, ここでプログラムの形式的な表記方法を定義準備し, 各世代で共通となるアルゴリズムをまとめておく．

Definition 1

- まず, 入力プログラム $PROG$は, グローバルプログラム G もしくは, ローカルプログラム Lに分類でき, グローバルプログラム G は, グローバル構文 (G_i) と並列制御構文 (P_i) の列として構成されると仮定する．

- 並列制御構文 P は方向属性をもち, それを $P.dir$ と表記し $\{X, Y, Z,$不定$\}$ のいずれかをとるものとする．ただし, 不定の状態とは X, Y, Zいずれの状態をもとり得ることを意味する．P の内部に複数の方向属性が混在するものは, 不適切なものとする．

- P 内で定義, 参照される変数の集合をそれぞれ $P.DEF$, $P.USE$ と表記することとする．

- 並列制御構文, グローバル構文 (以下 B と表記する) 中に出現する数式表現 $E(:= \{expr \in B\})$ に対して, 方向属性, 配列次元, インデックス変数名をそれぞれ $E.dir$, $E.dim$, $E.indexX$ と表記する．

- 基本ブロックは深さ優先探索によって, 一意にインデックス n が振られる．それを用いて, 入力プログラムは $PROG$={ $B(1), B(2), ..., B(|PROG|)$ } とも表すこととする．
  ここで, $|PROG|$ は, $PROG$ 中の基本ブロックの数を表す．

PDO ブロックの方向属性決定アルゴリズム

ADETRAN4 では, 各 PDO ブロック内での変数の方向属性は PDO ブロック内の変数表記によって一意 (もしくは不定状態) にならなくてはならない．PDO ブロック内の方向属性を決定するアルゴリズムを次に示す．このアルゴリズム中で, 数式 (E) の属性を決定するアルゴリズムは属性変換文法によって再帰的に定義されるルールに従ってボトムアップ式に決まるとする．ここで, $E.dir, E.dim, E.indexX$ はそれぞれ, 数式 E の方向属性, 次元, X 番目のインデックスを表す．

このアルゴリズムによって、各 PDO ブロックの属性は次の 8 種類, $\{X, Y, Z, X \cup Y, Y \cup Z, Z \cup X, X \cup Y \cup Z, \phi\}$ のいずれかに決まる．このとき, ϕ は不適切状態を表し、 X,Y,Z 以外は不定状態を表す．

$B.dir := \bigcap_{E \in B} E.dir$　(B：基本ブロック, E：B 内に出現する数式)

ただし 数式 E の方向属性 $E.dir$ は以下の翻訳規則に従って決定される:

$E \longrightarrow E_1 \oplus E_2$　(ここで $\oplus$ は任意の演算子(単項, 3項以上の演算も同様である))

{ $E.dir := E_1.dir \cap E_2.dir$ }

$E \longrightarrow < array >$

{　$E.dir := \phi$

if($E.dim$==2D)**then**

if($E.index2 \in <$ *pdo_ctr* $>$)$E.dir \cup = X$

if($E.index1 \in <$ *pdo_ctr* $>$)$E.dir \cup = Y$

else if($E.dim$==3D)**then**

if($E.index2 \in <$ *pdo_ctr* $>$ & $E.index3 \in <$ *pdo_ctr* $>$)$E.dir \cup = X$

if($E.index3 \in <$ *pdo_ctr* $>$ & $E.index1 \in <$ *pdo_ctr* $>$)$E.dir \cup = Y$

if($E.index1 \in <$ *pdo_ctr* $>$ & $E.index2 \in <$ *pdo_ctr* $>$)$E.dir \cup = Z$

endif

}

$E \longrightarrow < vector > | < scalar > | < const >$

{ $E.dir := X \cup Y \cup Z$ }

図 2.10: PDO ブロック B 内の変数の方向属性を決定するアルゴリズム

データ再分散 (PASS) 解析アルゴリズム

ADETRAN4 における方向属性は, VPP-Fortran やメッセージパッシングモデルにおけるデータ分割における分割位置の指定に対応する．ここで, ADETRAN4 でデータ分割の軸の位置が変更されることは, データのインデックス位置をローテションさせてインデックスと物理データの連続性を保証することを意味する.

的確さを期するために, HPF(High Performance Fortran[17, 18]) を用いて説明することとする．仮に HPF で 2 次元配列 A(I,J) が (*,CYCLIC) で分割されていた場合に, サブルーチンの呼び出し側で call sub(a(1,1)) とし, 受け側で変数を b(i) の形で受ける様なプログラムがあったとする．変数 b(1:n) に a(1:n,1) がマッピングされることは HPF の文法上保証される事項である．しかし, (CYCLIC,*) への再分散がなされた場合, これは ADETRAN4 で方向属性の変化を意味するが, call sub(a(1,1)) を b(i) で受けた場合の main:a→sub:b のマッピングは b(1:n) ≡ a(1:1+n*P:P) を保証するものでしかない (もちろん配列の継承方法にも依存するが, default ではこの様になる). ADETRAN4 では, 方向属性の変更に伴いデータの連続性がその方向属性を規定

するインデックス上で保証されるのである. つまり, HPF の様な再分散を行うことに加えて, 配列の格納方法における転置, または 3 次元以上であれば回転操作を同時に行うことを意味している.

この様に, ADETRAN4 のメモリモデル, つまり方向属性の異なる配列 (セグメント配列) を分散メモリ計算機上でエミュレートするためには, それぞれの方向属性の配列データとしてメモリ空間上に確保し, データ再分散が施された時点で適切な回転操作を伴うデータの転送を行い, プログラムの表記上のデータとメモリ空間上のデータの一致を図る必要が生じる.

ここで, 「適切な位置にデータ再分散を行う」ために適切な位置をプログラム解析により決定する必要が発生する. このデータ再分散解析を, ADETRAN での転送文 (PASS 構文) 挿入問題と対応づけることができることから, 以下 **PASS 挿入問題**と呼ぶこととし, データ再分散を **PASS** と呼ぶこととする.

PASS の検出方法は個々の PDO ブロック内で参照されるセグメント配列と定義されるセグメント配列に対するデータフロー問題の解として得られる. PASS の検出ならびに挿入位置決定問題を定式化するために, [54] の記法を参考に幾つかの用語を定義する.

Definition 2

1. 文 sが *PDO*ブロック nにあることを $s \in n$, 文 sでセグメント配列 x が定義されることを $s(x :=)$, また sで x が使用されていることを $s(:= x)$ と表す. さらに文 sの実行が文 s'の実行後であるとき $s' < s$ と表す.

2. *succ(n): PDO*ブロック n の後継者の集合.
   ここでブロック n_2 が n_1 の後継者であるとは, n_1 の最後の文から n_2 の最初の文へ到達する経路が存在することをいう.

3. *pred(n): PDO*ブロック n の先行者の集合.
   ここでブロック n_1 が n_2 の先行者であるとは, n_1 の最後の文から n_2 の最初の文へ到達する経路が存在することをいう.

Definition 3

1. *DEF(n): PDO*ブロック n で定義されるセグメント配列の集合
   $DEF(n) := \{x | s(x :=) \in n\}$

2. *USE(n): PDO*ブロック n で参照されるセグメント配列の集合
   $USE(n) := \{x | s(:= x) \in n \wedge (\forall s' < s \wedge s' \in n \Rightarrow s'(x :\neq))\}$

3. *DIR(n): PDO*ブロック n の方向属性
   $DIR(n) \in \{X, Y, Z, \emptyset\} (:= \mathcal{D})$

4. *Live_In(n):* n の入り口で *dir* 方向に関して生きているセグメント配列の集合
   $Live\_In(n) := \{(x, dir) | dir \in \mathcal{D} \wedge$ n の入口から x が *dir* 方向で使用される n' へ到達する任意の *path* 上に x の定義がない $\}$

5. *Live_Out(n):* n の出口で dir 方向に関して生きているセグメント配列の集合
   Live_Out(n):=$\{(x, dir)|dir \in \mathcal{D} \wedge$ n の出口から x が dir 方向で使用される n' へ到達する任意の $path$ 上に x の定義がない $\}$

6. *Reach_In(n):* n の入口に到達する可能性のある dir 方向のセグメント配列の集合
   Reach_In(n):=$\{(x, dir)|dir \in \mathcal{D} \wedge$ n の入口に達する $path$ 上に x の定義がある $\}$

7. *Reach_Out(n):* n の出口に到達する可能性のある dir 方向のセグメント配列の集合
   Reach_Out(n):=$\{(x, dir)|dir \in \mathcal{D} \wedge$ n の出口に達する $path$ 上に x の定義がある $\}$

この記法を用いて, 生存セグメント配列に関するデータフロー方程式は次の様になる.

Definition 4 (生存セグメント配列)

1. $Live\_Out(n) = \bigcup_{n' \in succ(n)} Live\_In(n')$
2. $Live\_In(n) = (Use(n), Dir(n)) \cup (Live\_Out(n) - (DEF(n), *))$

また, 定義到達セグメント配列に関するデータフロー方程式も次の様になる.

Definition 5 (定義到達セグメント配列)

1. $Reach\_In(n) = \bigcap_{n' \in pred(n)} Reach\_Out(n')$
2. $Reach\_Out(n) = (Def(n), Dir(n)) \cup (Reach\_In(n) - (DEF(n), *))$

上式によって求められた Live_(In/Out), Reach_(In/Out) に対して,

1. $PASS\_In(n) := \{(x, d_1, d_2)|(x, d_1) \in Live\_In(n) \wedge (x, d_2) \in Reach\_In(n)\}$
2. $PASS\_Out(n) := \{(x, d_1, d_2)|(x, d_1) \in Live\_Out(n) \wedge (x, d_2) \in Reach\_Out(n)\}$

を計算し, 更に $n \rightarrow n'$ と path がなっていた場合に $PASS\_Out(n) \cap PASS\_In(n')$ を求めることで, ある変数 (x) がどの方向属性で定義されて (d_1) その後にどの方向属性で参照されるか (d_2) が決定できるとともに, path 上にその情報が残ることとなる. この情報を基に, 定義場所から最も近い最外側ループの位置に PASS を置くことが決定できれば, PASS の発行回数を最小限にでき, かつ参照されない方向属性への PASS も行なわれない. この一連の解析アルゴリズムを図 2.12に示す.

procedure analysis_of_PASS_SYNC:

solve LV解 ⇒ { Live_In() & Live_Out() }

set Live_Out ← ϕ, Live_Out ← ϕ.

repeat until stabilized

for ∀ i, Live_Out(i)= $\bigcup_{j \in \mathrm{succ(i)}}$ Live_In(j),

Live_In(i)=(Use(i),Dir(i)) ∪ (Live_Out(n)−(Def(i),*))

end repeat

solve RD解 ⇒ { Reach_In() & Reach_Out() }

set Reach_Out ← $\bigcup VAR$, Reach_Out ← $\bigcup VAR$.

repeat until stabilized

for ∀ i, Reach_In(i)= $\bigcap_{j \in \mathrm{pred(i)}}$ Reach_Out(j),

Reach_Out(i)=(Def(i),Dir(i)) ∪ (Reach_In(n)−(Def(i),*))

end repeat

solve PASS解 ⇒ { Pass_In() & Pass_Out() }

find most outer path in { Pass_In ∩ Pass_Out() }

insert *PASS statement* into the **nearest** points and

SYNC statement into the **farther** points.

end procedure

図 2.11: PASS,SYNC挿入アルゴリズム

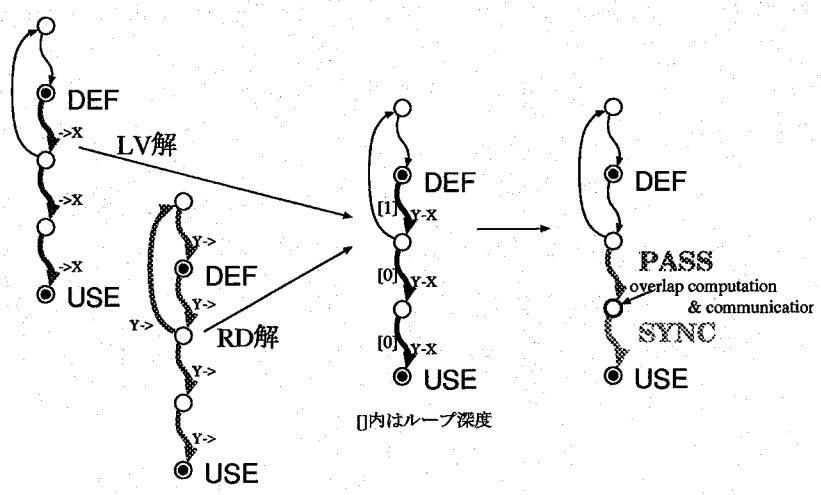


図 2.12: PASS,SYNC挿入アルゴリズムにおけるデータフロー解のチャート図 (LV解はLive_In∩Live_Out, RD解はReach_In∩Reach_Outを図示している)

グローバル変数解析

分散メモリ上での ADETRAN4 の実装において, スカラ変数の扱いは先に示した様に moly 変数であるとともに各プロセッサ毎に値を一致させる構文上の取り決めは UNIFY 文以外にない. しかしながら, 幾つかの制御構文にはスカラ変数がその制御カウンタとして利用されており, GDO などの G 構文や, PDO 構文などではそれを実行するすべてのプロセッサ間で値が一致したグローバル変数でなくては正しい動作 (全プロセッサの coherent な動作) が保証されない. P 構文の内部では個々の論理プロセッサの動作は一致する必要がないので, 制御カウンタに使用されているスカラ変数の一致を保証する操作 (unification) を行う必要が生じる. これは, L 構文以外の制御カウンタをグローバル変数化する問題に帰着できる. ただし, あくまでもスカラ変数は moly 変数であり UNIFY 構文以外での unification は行わないものとする.

このとき, 一般のプログラムでは静的にグローバルとして確定する変数が多く利用されるため, 制御構文全てに unification を行う必要はない. 静的解析によって, ローカル変数になりうるものを利用する部分にのみ unification を行うことで, 通信を伴う無駄な処理を削減でき性能面での優位性を保てる.

このように, 本節では静的解析によるグローバル変数解析について述べる. グローバル変数解析も前節に示した, PASS 挿入問題と同様にデータフロー問題を解くことによって決定される.

以下, データフロー方程式を導出するための幾つかの諸定義を行うとともにグローバル変数解析のアルゴリズムを示す.

Definition 6

1. *NULL_In(n)*:=$\{ x \mid n$ の入口に達する任意の *path* 上に x の定義がない $\}$
2. *NULL_Out(n)*:=$\{ x \mid n$ の出口に達する任意の *path* 上に x の定義がない $\}$
3. *GLOBAL_In(n)*:=$\{ x \mid n$ の入口に達する任意の *path* 上に x の *GLOBAL* な定義がある $\}$
4. *GLOBAL_Out(n)*:=$\{ x \mid n$ の出口に達する任意の *path* 上に x の *GLOBAL* な定義がある $\}$

Definition 7 (NULL 変数解析)

1. $NULL\_In(n) := \bigcup_{n' \in pred(n)} NULL\_Out(n')$
2. $NULL\_Out(n) := NULL\_In(n) - DEF(n)$

Definition 8 (GLOBAL 変数解析)

1. $GLOBAL\_In(n) := \bigcap_{n' \in pred(n)} GLOBAL\_Out(n')$
2. $GLOBAL\_Out(n) := GLOBAL\_In(n) \cup GLOBALDEF(n) - LOCALDEF(n) - NULL\_Out(n)$

Definition 9 (GLOBALDEF 解析)

手続き GLOBALDEF, LOCALDEF は次のアルゴリズムで定義される.

1. **procedure** GLOBALDEF(n):

```
tmp := φ
forall ( s ∈ B(n) )
  gvar_tmp:= GLOBAL_In(n) ∪ tmp ∪ Const
  if ( any_elements_of(s) ∈ arg_of_FUNCTION ) return φ
  switch ( s )
  case 代入文
    if ( not all_elements_of(RHS) ∈ gvar_tmp ) return φ
    tmp∪ = LHS
  case DO文
    if ( not all_elements_of(DO_START) ∈ gvar_tmp
         ∨ not all_elements_of(DO_END) ∈ gvar_tmp
         ∨ not all_elements_of(DO_STRIDE) ∈ gvar_tmp ) return φ
    tmp∪= DO_CTR
  case IF文
    if ( not all_elements_of(EXPR) ∈ gvar_tmp ) return φ
  default
    if ( s involves definitive operation ) then
      forall(expr ≺ s )   /* where '≺' means 'elements of' */
        if(not all_elements_of_(expr) ∈ gvar_tmp ) return φ
        add defined variables to tmp
      end forall
    endif
  end switch
end forall
return tmp
end procedure
```

2. *LOCALDEF(n):=DEF(n)−GLOBALDEF(n)*

2.3 第一世代：並列計算機 ADENART での実装

第一世代においては, 入力ソースの論理的なモデルが ADETRAN4 の想定する交互方向に基づくものであるため, 翻訳系は ADETRAN がサポートしない言語部分を処理可能とする構文ならびにデータ構造の変換を行う. 研究 [43] では, 並列言語 ADETRAN4 で書かれたプログラムを並列計算機 ADENART で実行させるための試みとして, ADETRAN4 から ADETRAN へ翻訳するプリプロセッサ adeconv が開発された.

ADETRAN4 と ADETRAN の仕様上の主な違いは表 2.1のとおりである. adeconv では, メモリモデルの差異を, 使用する方向属性毎にデータ領域を確保し, PASS 挿入問題によって自動的にデータ再分散を行いデータ整合をとる処理によって解決している.

またさらに, ADETRAN4 で追加された拡張配列をエミュレートする機能は配列の一次元化によって実現する. ただし, セグメント配列の拡張次元を主次元の方向属性インデックスの一次元化で実現する場合, PASS 文で制約が生じる (ADETRAN では PASS 文によって再分散可能な範囲は REGION 文によって指定された領域に限られている). それを解決するために図 2.13に示す窓を使う方法で解決している. なお 3 次元の PCAST 文は, アセンブラによるランタイムライブラリを作成することで対応した.

adeconv が行う翻訳処理は次の様にまとめられる.

(解析 0) 字句, 構文解析

(解析 1) USE, DEF ならびにセグメント配列方向属性の解析を行う.

(解析 2) PASS 挿入問題を解き, 必要部分に PASS 文を挿入する.

(コード生成 1) 拡張配列部分の 1 次元化を行い, 方向属性毎にセグメント配列の宣言を行う.

(コード生成 2) ADETRAN4 基本構文 (PDO, PCAST2 次元, GDO, GIF, IFALL, IFANY 文) のコード生成を行う. 但し, コード生成の際に, 拡張次元部分を対応する方向属性部分に 1 次元化する.

(コード生成 3) 拡張セグメント配列に関する PASS 文の展開を行う. 3 次元 PCAST 文用関数を call 生成する.

表 2.1: ADETRAN4 と ADETRAN の主な違い

	ADETRAN4	ADETRAN
想定するメモリモデル	分散共有	分散
セグメント配列の宣言	使用する方向に無関係	使用する方向毎に必要
方向属性の表現	不要	必要 (スラッシュ対にて表現)
データ再分散 (PASS 文)	不要	必要 (PASS 文にて記述)
拡張次元	可	不可
PCAST 文	2,3 次元可 (含拡張次元)	2 次元のみ可

- 入力ソース (PASS 文部分のみ)

```
pass i=n1,n2, j=m1,m2, k=l1,l2
   a(/i/,j)(k)=a(i,/j/)(k)
pend
```

- 出力ソース (対応する部分のみ)

```
gdo k=l1,l2
  pdo i=n1,n2
    do j=m1,m2
      buf(/i/,j)=a(/i/,j+(k-1)*(m2-m1+1))
    enddo
  pend
  pass i=n1,n2, j=m1,m2
    buf(i,/j/)=buf(/i/,j)
  pend
  pdo j=m1,m2
    do i=n1,n2
      a(i+(k-1)*(n2-n1+1),/j/)=buf(i,/j/)
    enddo
  pend
genddo
```

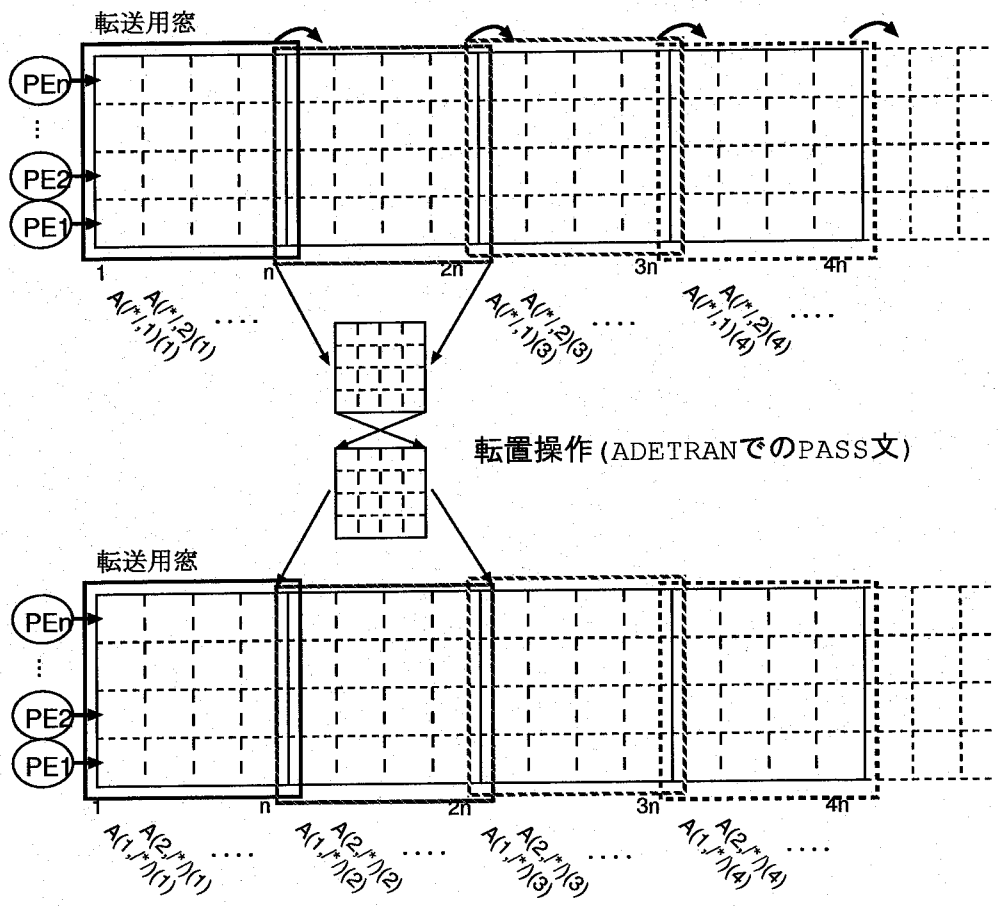


図 2.13: 拡張セグメント配列実装と PASS 文実行のための転送用窓の使用 (上記左側プログラムが PASS 挿入解析によって中間言語中に挿入された PASS 文, 上記右側プログラムが ADETRAN へ変換結果)

以上の様な仕様と実装方針の下で, 翻訳系 adeconv は左記の ADETRAN4 プログラムを右にある様なプログラムに変換する. ここで変数 buf が PASS 文における REGION 文の制限を緩和するための転送領域である.

- 入力ソース

```
real a(n1,n2,n3)(n4)
real v(n2)(n3)
pdo  i2=1,n2, i3=1,n3
   do i4=1,n4
      do i1=1,n1
         a(i1,i2,i3)(i4)=...
      enddo
   enddo
pend
pdo  i3=1,n3, i1=1,n1
   do i4=1,n4
      do i2=1,n2
         ... = a(i1,i2,i3)(i4)
      enddo
   enddo
pend
pcast  i2=1,n2, i3=1,n3
   v(i2)(i3) = a(1,i2,i3)(1)
pend
     ...
```

- 出力ソース

```
real a(n1*n4,/n2,n3/),a(n1/,n2*n4,/n3)
real v(n2*n3)
real buf(n1,/n2,n3/),buf(n1/,n2,/n3)
pdo  i2=1,n2, i3=1,n3
   do i4=1,n4
      do i1=1,n1
         a(i1+(i4-1)*n1,/i2,i3/)=...
      enddo
   enddo
pend
gdo i4=1,n4
   pdo  i2=1,n2, i3=1,n3
      do  i1=1,n1
         j=i1+(i4-1)*n1
         buf(i1,/i2,i3/)=a(j,/i2,i3/)
      enddo
   pend
   pass  i1=1,n1, i2=1,n2, i3=1,n3
      buf(i1/,i2,/i3)=buf(i1,/i2,i3/)
   pend
   pdo  i3=1,n3, i1=1,n1
      do  i2=1,n2
         j=i2+(i4-1)*n2
         a(i1/,j,/i3)=buf(i1/,i2,/i3)
      enddo
   pend
genddo
pdo  i3=1,n3, i1=1,n1
   do i4=1,n4
      do i2=1,n2
         ... = a(i1/,i2+(i4-1)*n2,/i3)
      enddo
   enddo
pend
j=1+(1-1)*n1
call pcast3(v,a(j,/,/),1,n2,1,n3)
   ...
```

図 2.14: adeconv 変換例

2.4 第二世代：ベクトル並列計算機 VPP500 での実装

ベクトル並列処理は, 従来のプロセッサ単位の並列処理に加えてベクトルプロセッサ内処理での並列処理が加わる処理である. 並列処理方式 ADEPS は, 2 次元ならびに 3 次元配列の計算を 1 次元配列単位の処理に分ける. このとき, 1 次元配列の集合としての演算はベクトル処理可能であり, 3 次元配列の場合, 3 次元インデックスの 1 つをベクトル処理, 1 つを並列処理, もう一つを逐次処理にもベクトル処理にも割り当てられる. 2 次元配列の場合は, 一つを並列処理に, もう一つのインデックスをベクトル処理できる. つまり, ADEPS の処理方式の中にはベクトル処理を内在していると言える. 研究 [48, 44, 45] では, ベクトル並列型計算機上での ADETRAN4 処理系の実装を行うとともに, それを用いた応用計算を行い評価を行っている.

研究 [48, 44, 45] では, ADETRAN4 コンパイラを VPP500 のオブジェクトコードを生成するネイティブコンパイラとしてではなく ADETRAN4 言語から並列計算機 VPP500 で提供されている並列化言語 VPP-Fortran[49, 50] へ変換するプリプロセッサとして実装した.

なお, 本処理系の翻訳処理は 23頁で示した adeconv の処理の流れと同様である. adeconv とはターゲットマシンが異なるため, コード生成部分 (1~3) が異なることとなる. 以下, コード生成部分に関して示していく.

2.4.1 データ分散と並列実行

ADETRAN4 における方向属性は VPP-Fortran では図 2.15のように方向属性そしてグローバル・ローカルを識別するためのタグを変数名に付加し, 方向属性を持つセグメント配列を管理する. 実行時には異なる方向属性分の配列が必要となる. ここで配列の添字がシフトしているのは, シフトを行わない場合に第一インデックス以外がベクトル化対象となるため, メモリバンクアクセスの軋轢によって起因するベクトル化率や実行速度のアンバランスをなくすことを顧慮した結果である.

変数名の管理並びに VPP-FORTRAN での配列へのマッピング (インデックスの位置) は次に示すアルゴリズム (図 2.16) で行っている. ただし, 3 次元セグメント配列A が PDO ブロック B で参照・更新される場合を示している. 2 次元セグメント配列も同様に図 2.17で行われる. またこの命名法の他にグローバル分割配列にマッピングするためにg?_ で始まる配列の宣言を行い EQUIVALENCE 宣言する必要がある. この変数の必要性は VPP-FORTRAN での転送命令の文法上の制約から来るものであり, 転送命令時にのみ利用するものである.

分割軸の位置はプロセッサ数を可変に実行できることを考慮し, 第 3 インデックスにそれを割り当てている. またベクトル実行効率の最も良くするために, 第 1 インデックスにベクトル実行の添字部分を割り振り, データは VPP-FORTRAN の制約からプロセッサ数を固定した形で, CYCLIC でプロセッサへマッピングすることとした.

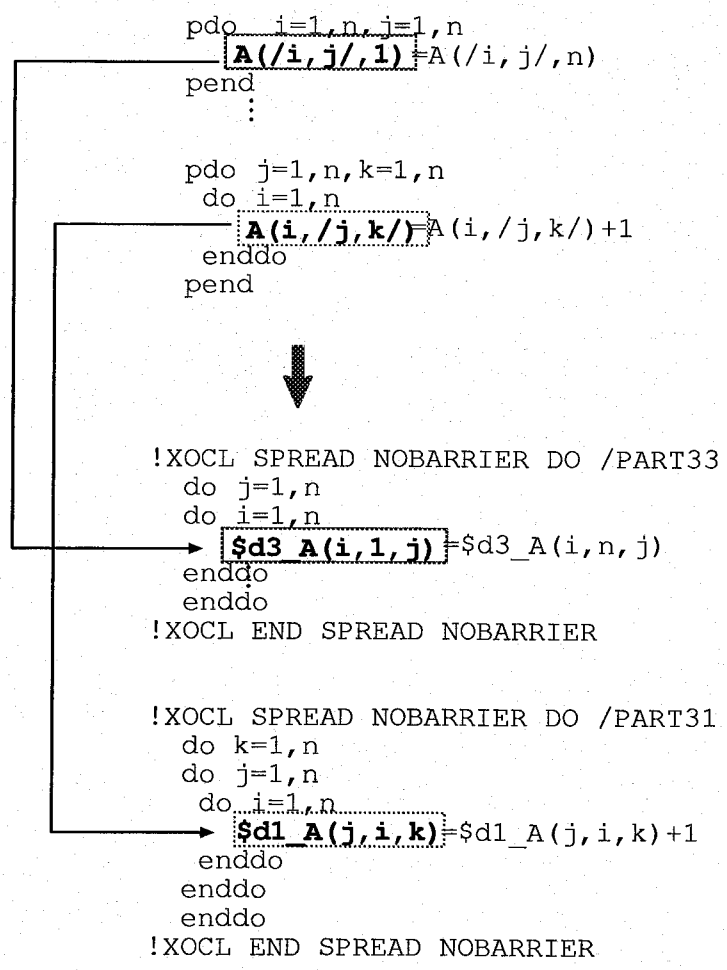


図 2.15: データ分散と並列実行部分の実装

$A(i1, i2, i3) \Rightarrow$

if ($B.dir == X$) **then**

nameof(A) := `$d1_A` & `$g1_A`, $< index >:= (i2, i1, i3)$,

mapping `$d1_A` **to** (*,*,/CYCLIC[ローカル分割配列])

mapping `$g1_A` **to** (*,*,/CYCLIC[グローバル分割配列])

endif

if ($B.dir == Y$) **then**

nameof(A) := `$d2_A` & `$g2_A`, $< index >:= (i3, i2, i1)$,

mapping `$d2_A` **to** (*,*,/CYCLIC[ローカル分割配列])

mapping `$g2_A` **to** (*,*,/CYCLIC[グローバル分割配列])

endif

if ($B.dir == Z$) **then**

nameof(A) := `$d3_A` & `$g3_A`, $< index >:= (i1, i3, i2)$,

mapping `$d3_A` **to** (*,*,/CYCLIC[ローカル分割配列])

mapping `$g3_A` **to** (*,*,/CYCLIC[グローバル分割配列])

endif

図 2.16: 3 次元セグメント配列 A の命名法とマッピング

$A(i1, i2) \Rightarrow$

if ($B.dir == X$) **then**

nameof(A) := `$d1_A` & `$g1_A`, $< index >:= (i1, i2)$,

mapping `$d1_A` **to** (*,/CYCLIC[ローカル分割配列])

mapping `$g1_A` **to** (*,/CYCLIC[グローバル分割配列])

endif

if ($B.dir == Y$) **then**

nameof(A) := `$d2_A` & `$g2_A`, $< index >:= (i2, i1)$,

mapping `$d2_A` **to** (*,/CYCLIC[ローカル分割配列])

mapping `$g2_A` **to** (*,/CYCLIC[グローバル分割配列])

endif

図 2.17: 2 次元セグメント配列 A の命名法とマッピング

2.4.2 データ転送

分散メモリ計算機上でのADETRAN4の実装を行う場合, データの転送方式は大きく分けると次の3種類に分類される.

1. **PASS** 方向属性の変化 (分割軸の循環変更)
2. **BCAST** あるプロセッサから全プロセッサへのデータ放送
3. **PCAST** 全プロセッサの個々のデータを他プロセッサに放送

1. の転送方式をPASS, 2. の転送方式をBCAST, 3. の転送方式をPCASTと呼ぶ. PASSは, 第2.2節での準備でも述べているように異なる方向属性のデータとの間で整合性を保たせるためのデータ再分散である. BCASTは一般にブロードキャストと呼ばれる集合通信方式である. PCASTはgather-all(MPIでは`MPI_Allgather`) と呼ばれる集合通信方式の特殊な場合である.

1) PASS

VPP-FORTRANでは, PASS形式のデータ転送は`SPREAD MOVE`文と`MOVEWAIT`文とで実現される. PASS挿入問題の解 (頁17) に対して, SPREAD MOVE構文とMOVEWAIT構文を挿入する処理を行う.

ここで, SPREAD MOVE構文とMOVEWAIT構文を十分に離して配置することで, 通信と計算のオーバーラップを実現することができる.

2) BCAST & 3) PCAST

BCASTならびにPCASTは, PCAST構文の右辺に現われるセグメント配列の方向属性によってどちらの転送を行うかが選択される. この選択は次に示すように, 構文解析の段階で判断できるため, コンパイラが適当な転送方式を決定し, 転送コードを出力する.

次のプログラムにあるように, スラッシュ対によって指定された分割軸の位置がpcast構文のカウンタに無い場合は, j番目のプロセッサのセグメント配列aの要素1からnまでを, 全ての物理プロセッサ上のセグメントベクトルbに放送する. したがって, この様な記述の場合BASCTを選択する.

```
pcast  i=1,n
   b(i)=a(i,/j/)
pend
```

また, 次のプログラムの様にスラッシュ対によって指定された分割軸の位置がpcast構文のカウンタに等しい場合は, 各i番目の論理プロセッサ上のセグメント配列aのj番目の要素を, 全ての物理プロセッサ上のセグメントベクトルbのi番目の要素の放送する. したがって, この様な記述の場合PCASTを選択する.

```
pcast  i=1,n
    b(i)=a(/i/,j)
pend
```

VPP-FORTRAN には BCAST に相当する構文として, BROADCAST 文があるため, ADETRAN4 ソースコードから VPP-FORTRAN コードを生成するプリプロセッサではそれを用いて BCAST を実現する.

PCAST の実装方法は, 次の 3 つが現実的な手法として考察することができる.

(a) ループを分割しない SPREAD MOVE 文によってブロードキャストを行う方法

(b) プロセッサ個々のデータを一旦 packing し, 全てのプロセッサを放送元とする BROADCAST 文を一通り行う. その後で配列要素の並び替えを行う方法.

(c) SPREAD MOVE 文で一旦あるプロセッサ上にデータを集約し, その後 BROADCAST 文で放送する方法がある.

これら 3 種類の方法で PCAST を行った場合の時間を測定した結果が図 2.18である. ここで (a) は実験時環境の制約からデータ長 1000 までしか実現できなかった. また (b),(c) についても利用可能メモリの制限から, データ長 10000 までしか実測することができなかった. 比較として BCAST の転送を行った場合の転送時間を (d) に, そして最大転送レート 400MB/s から算出される転送時間の下限値を (e) としてプロットした.

この結果から (e)<(d)<(b)<(a)<(c) の順となることが分かった. データ長 10000 では (b) と (c) はほぼ同程度の転送性能となるが, 実際に利用するデータは 10000 以下程度であることから, ADETRAN4 プリプロセッサでは (b) による PCAST の実装が適当であると判断できる. 実際の処理系の実装では (b) を採用することとした.

実際, (b) の実装には図 2.19にあるような関数をプログラム中に登場する PCAST 構文個々に生成し, 関数を呼び出す方式をとっている. この方式をとった理由は, VPP-Fortran の仕様において BROADCAST 構文の引数に可変な配列もしくは部分配列要素を指定できないためである.

2.4.3 サブプログラム

ホストプログラムからのみ呼び出される **G** サブルーチンは VPP-FORTRAN で並列処理の区間を指定するPARALLEL REGION~END PARALLEL 構文を含む並列化手続きとして変換する. **G** サブコルーチンは並列処理中に並列処理に携わるすべてのプロセッサ群がそれに制御を移行するため VPP FORTRAN でのスプレッド手続きとした. 今回これら手続き間の変数受渡しは common ブロックを用いたもののみで行ないプロセッサ形状は固定とした. また **L** サブルーチン・**L** 関数は PURE な手続きとして変換し, 変数の受渡しは引数で指定する方法とセグメントベクトルのみを含む common ブロックによる方法の二つを可能とした. なお PURE という単語そのものは HPF[17] で FORALL 構文で利用可能な副作用のない関数として新たな概念が定義されたものであり, FORALL と PURE は FORTRAN95[1] で新規構文として認定されたものである.

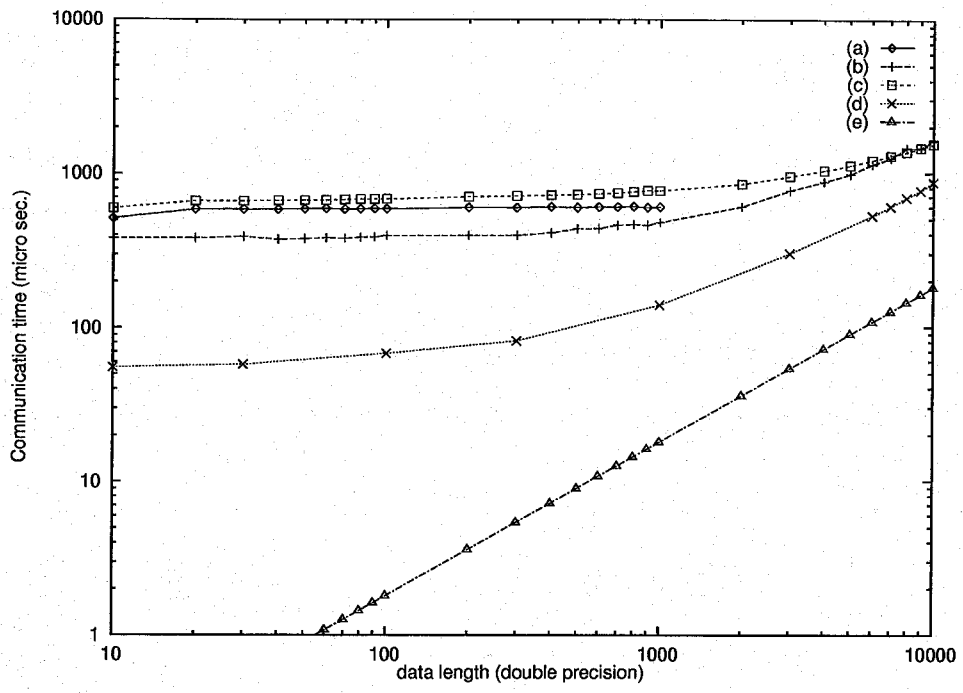


図 2.18: 各 PCAST の実装方法で要した転送時間

```
      subroutine $$_pcast00001(b)
*XOCL PROCESSOR GSUB(4)
*XOCL SUBPROCESSOR GSUBCO(4)=GSUB(1:4)
      double precision  b0(250), b1(250), b2(250), b3(250)
      double precision  b(250), bb(1000)
      equivalence (bb(1),b0(1)), (bb(251),b1(1)), (bb(501),b2(1))
      equivalence (bb(751),b3(1))
      do i=1,250
         bb((IDVPROC()-1)*250+i)=b(i)
      enddo
*XOCL BROADCAST (b0) (1.EQ.IDVPROC())
*XOCL BROADCAST (b1) (2.EQ.IDVPROC())
*XOCL BROADCAST (b2) (3.EQ.IDVPROC())
*XOCL BROADCAST (b3) (4.EQ.IDVPROC())
*VOCL LOOP, UNROLL, DISJOINT(b,bb)
      do i=1,250
      do j=1,4
         b((i-1)*4+j) = bb((j-1)*250+i)
      enddo
      enddo
      return
      end
```

図 2.19: VPP-Fortran での PCAST 構文の実装: `pcast i=1,1000; b(i)=a(/i/,1); pend` を 4 プロセッサで実行するための関数例

2.4.4 デバックとチューニング

ADETRAN4プリプロセッサはEWS上でクロスコンパイラの形式で開発をした．プリプロセス実行はEWS上で行うことになる．ADETRAN4からVPP-Fortranに変換したプログラムの利点として，コンパイラ指示子を完全なコメントとしてみなしても，逐次実行において殆んど等価なFORTRAN77プログラムとなる点にある [49]．この利点を活かしVPPのフロントエンドであるUXPのみならず標準的なEWS上のFORTRAN77コンパイラや豊富なデバッグツールを用いることができる．

VPPの性能を十分に発揮するには，個々の単体ベクトルプロセッサでのベクトル化が十分になされなければいけない．ユーザーはプリプロセッサで変換されたVPP-Fortranプログラムを通じて，VPPコンパイラにベクトル化が促進されるようにベクトル計算機における一般的なチューニングを行なう．研究 [44] では次の方法が有効であった．

1. 計算をまとめたりDOループをタイトにし，ベクトルレジスタの利用効率をあげる．
2. 多次元配列の宣言範囲を変えバンク競合を減らす．
3. ベクトル化促進・抑制指示子 (VPP-Fortran用) を積極的に入れる．

上記のチューニングを行なうことで10%から50%の速度向上が見られる．

2.5 第三世代：MPI を用いた処理系実装

研究 [46] では, ADETRAN4 を通信ライブラリ (日立 SR2201 での EXPRESS ライブラリ) を用いた SPMD コードに変換する研究結果と, さらに, 事実上の業界標準となった MPI(Message Passing Interface)+Fortran90 に変換するプリプロセッサ開発について報告している. 本プリプロセッサ開発の結果, 標準通信ライブラリである MPI と Fortran90 が利用可能な殆んどの並列計算機上で ADETRAN4 が利用可能となった. 本節では, MPI を利用したメッセージパッシングコードへ変換するプリプロセッサ実装の説明を行う. なお, 本処理系も VPP500 への実装と同様に 23頁で示した adeconv とほぼ同様の構成をとる. 本節では, ターゲットマシンが異なること及び拡張された言語仕様部分に関して, 異なる実装部分を以下に示していく.

2.5.1 データ構造 と並列実行制御

MPI を利用したメッセージパッシングコードの実装においても, VPP-FORTRAN を用いた処理系実装と同様に, 方向属性を識別するためのタグを変数名に付加している. VPP500 での実装と同様に 3 次元セグメント配列については以下の様な命名規則とマッピングを行っている (図 2.20). 2 次元セグメント配列は変数の接頭語が `1?$` になった以外 VPP500 でのものと何ら変わらない. ここで, VPP-FORTRAN では SPREAD MOVE 構文によるデータ転送を行うためにグローバル変数とローカル変数の 2 種類を定義し区別をしていたが, メッセージパッシングモデルではその区別を行わない. ただし, 次節で述べる動的なメモリ管理のために接頭語 `p?$` で始まる別名を利用している.

$A(i1, i2, i3) \Rightarrow$
if ($B.dir == X$) **then**
  nameof(A) := `11$A`, $< index >:= (i1, i2, i3)$,
  mapping (*,*,proc_ID::proc_NO) → (*,*,1::1) at proc_ID :: ローカル配列
endif
if ($B.dir == Y$) **then**
  nameof(A) := `12$A`, $< index >:= (i2, i3, i1)$,
  mapping (*,proc_ID::proc_NO,*) → (*,*,1::1) at proc_ID :: ローカル配列
endif
if ($B.dir == Z$) **then**
  nameof(A) := `13$A`, $< index >:= (i3, i1, i2)$,
  mapping (proc_ID::proc_NO,*,*) → (*,*,1::1) at proc_ID :: ローカル配列
endif

図 2.20: 3 次元セグメント配列の命名法とマッピング

2.5.2 プロセッサ可変のためのプログラム構造とその実装

これまで, 述べてきた第1, 第2世代の処理系ではコンパイル時に利用するプロセッサ数を指定しなくてはならず, 実行時にそれを変更することはできなかった. これは, ターゲットとなる言語がプロセッサ数可変でなかったためである. VPP-Fortranでは, プロセッサグループ形状をサブルーチン間で継承可能とすることでサブプログラム間でのプロセッサ数可変は実現可能であるがローカル分割配列を利用する部分で制約を受けるために実現できなかった.

メッセージパッシングライブラリを用いる場合には, 配列ならびに変数の管理をはじめとして通信の順序などが全て利用者の責任の下, ほとんどの処理が可能となっている. 無論, これらの管理項目を正しく行えばプロセッサ数可変のプログラム生成は不可能でない.

第3世代の処理系では, これを実現するためにセグメント配列をFortran90が提供するallocatable属性配列として動的に確保する方法を採用している. セグメント配列は, 実行時のプロセッサ数に応じて各プロセッサ上に占めるデータ領域が変化するため, 図2.21にあるように宣言している. この方式の採用により, 実行時に使用するプロセッサ数を自由に変更できるプロセッサ数可変なコードが生成できるようになった. VPP-FORTRANでの処理系実装では, プロセッサ数はコンパイル時に指定した個数に固定されていたため, 使用するプロセッサ数を変更するには再コンパイルを行う必要があった. しかし, MPIを用いた処理系ではその必要はなくなった.

しかしながらFortran90で採用された動的配列は, 旧来のFortran配列とは性格が異なり, 配列の形状並びに大きさも含めて関数に引き渡すため動作不安定や性能劣化を起こすことが予備実装から明確となった. これを解決する策として, 多次元配列を1次元配列として領域だけを確保し, サブプログラムに整合配列として渡す従来のFORTRAN方式が有効である. つまり, 図2.21に示す様に(サブ)プログラムの構造として 配列を確保するレイヤ(HEADER部分), 整合配列として受け実際の計算を行うレイヤ(BODY部分)の2階層として構築することとした.

この2階層レイヤによるサブプログラムの実装により, BODY部分では次に挙げる項目が可能となった.

- 配列を従来のFortran77と同様に, 整合寸法配列として扱うことができる.
- 他言語とのインタフェースを行う際に, Fortran77で行っていたことと同様の扱いができる.
- コモンブロックにポインタを列挙することで, 実装上可変とならざるを得ない配列のプログラム間での共有化がコモン文を用いたままで実装することが可能となった.

これら項目を, 図2.22に図示した. 中央にあるSUB1$HEADERがサブプログラムSUB1の入り口に相当する部分であり, SUB1で使用するセグメント配列のポインターを管理する. SUB1が一旦callされると, SUB1$HEADERでセグメント配列(この場合AとB)の領域を確保し, BODY部分に引き渡す. BODY部分は, 従来のFortran77と同様の変数引き渡しを行っているので, 1次元配列として引数の配列を外部関数に(ADETRANでないFortranもしくはCなどに)渡しても問題はない.

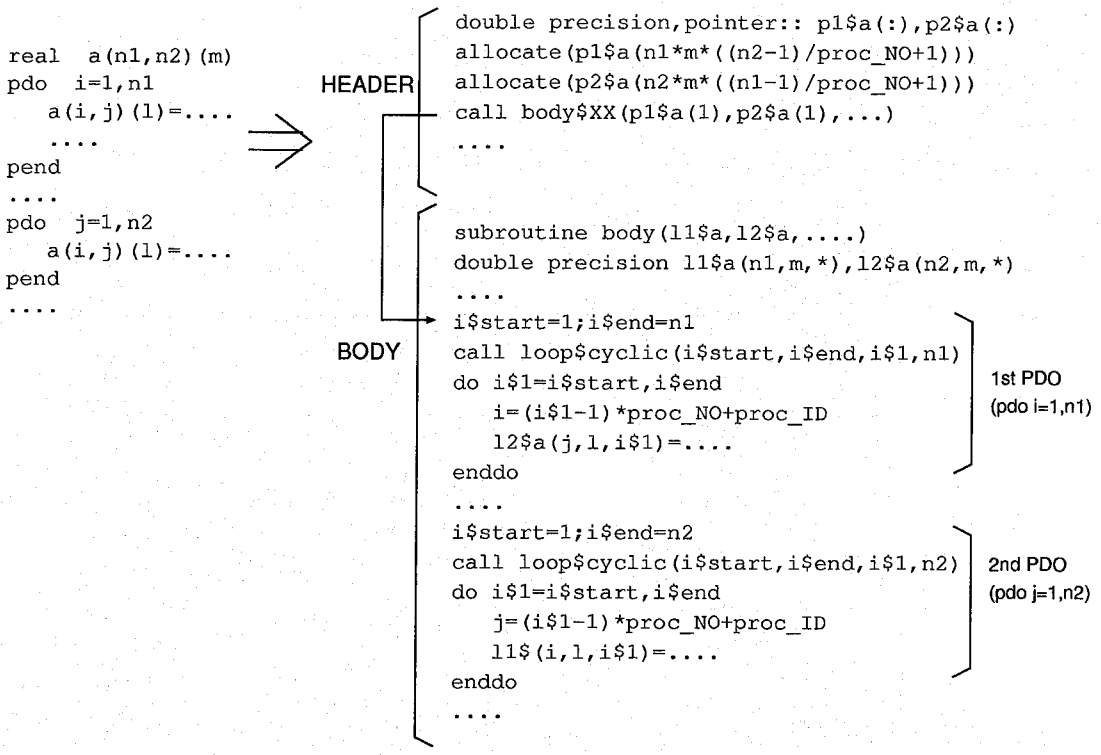


図 2.21: データの分散と並列実行制御部分の変換

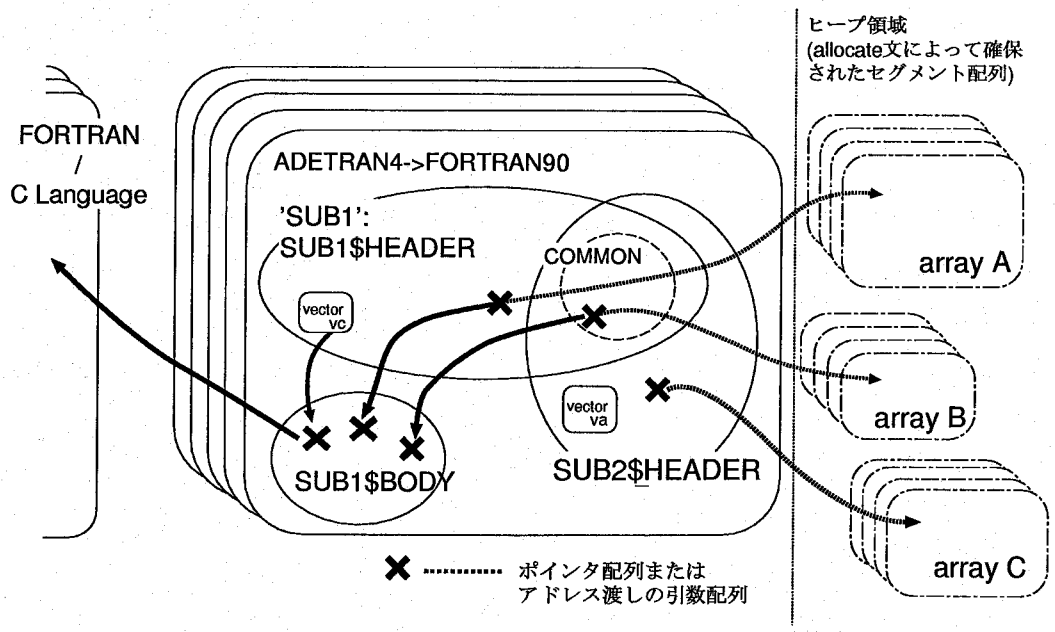


図 2.22: 可変プロセッサ実装におけるデータの分散とサブプログラム間インターフェース, セグメント配列 (A,B) は動的に確保され, ベクトル vc は静的に確保される. 変数群はすべて HEADER 内で 1 次元配列として確保され, BODY に渡される. また 配列 B は COMMON ブロック中にポインタがエントリされ他プログラム (SUB2) と共有される. BODY 部からは, FORTRAN77 と同様の外部言語インタフェイスが可能である.

1) 並列実行制御構文 PDO 文の実装

次に, 並列実行制御文 PDO 文の実装に関して触れる. 並列実行制御 PDO 文 は各プロセッサの担当部分を回す DO ループとして実装される. ただし, 個々のプロセッサ毎にループの回転数(担当範囲) が異なるため, ループの開始, ループの終了をあらかじめ決定しなくてはならない. 図2.21ではサブルーチン`loop$cyclic` において, 担当部分の計算を行っている. CYCLIC 分割を行った場合に, PDO 文の実行範囲が論理プロセッサ $(start, end)$ と指定されたときプロセッサ番号`proc_ID` が担当する領域 $(start', end')$ は次式で求められる. ここで, `proc_NO` は利用する総物理プロセッサ数を表す.

$$start' = \begin{cases} \lceil \frac{start}{proc\_NO} \rceil & proc\_ID \geq \mathrm{mod}(start, proc\_NO) \quad (2.5.1a) \\ \lceil \frac{start}{proc\_NO} \rceil - 1 & proc\_ID < \mathrm{mod}(start, proc\_NO) \quad (2.5.1b) \end{cases}$$

$$end' = \begin{cases} \lceil \frac{end}{proc\_NO} \rceil + 1 & proc\_ID > \mathrm{mod}(end, proc\_NO) \quad (2.5.2a) \\ \lceil \frac{end}{proc\_NO} \rceil & proc\_ID \leq \mathrm{mod}(end, proc\_NO) \quad (2.5.2b) \end{cases}$$

ここで, $(start, end)$ はグローバルである必要がある. これらの値がローカルで異なる場合には unification を行う必要が生じる. その可能性を判別する手段として, 21頁で述べたグローバル変数解析を用いて PDO 実行時前に非グローバルな場合に unification を発行する.

次に, 通信に関しての実装部分を示す. 第 2 世代 (VPP500) での実装で触れたように, ADETRAN4 では PASS,PCAST,BCAST の 3 種類の通信を実装する必要がある. それぞれ順に説明していくことにする.

2) PASS 文の実装

2 次元セグメント配列を例にして説明を行うことにする. 今, PASS 挿入解析によって`A(N1,/N2/)` から`A(/N1/,N2)` の間での PASS 文発行が必要であると判定されたとき, サイクリック分割において, プロセッサ i に格納されている`A(N1,/N2/)` の要素は次の通りである.

```
A(1,/i/),A(2,/i/)....,A(N1,/i/),A(1,/i+P/),....
```

ここで, P は実行時のプロセッサ数である. このとき,

```
A(j,/i/),A(j+P,/i/),...,A(j,/i+P/),A(j+P,/i+P/),...
```

がプロセッサ j へ転送されるべき配列要素となる. したがって, j に関するループを構成し, 順次, この配列要素をバッファ領域にパッキングしプロセッサ j へ送信し, 受信側のプロセッサ j で正しい要素位置に並び替える処理を行えば良い. PASS 文のアルゴリズムは図 2.23のようにまとめられる. MPI を用いた実装では, **send** に関数`MPI_Send`, **recieve** に関数`MPI_Recv` を用いる.

```
procedure PASS2D(source,dest)
    for  i=1..P
      pack [buff] ← { source(i,1),source(i+P,1),...,source(i,2),source(i+P,2),....}
      send [buff] to processor(i)
    endfor
    for  i=1..P
      recieve [buff] from processor(i)
      unpack [buff] → { dest(i,1),dest(i+P,1),...,dest(i,2),dest(i+P,2),....}
    endfor
end procedure
```

図 2.23: メッセージパッシングモデルでの PASS 文アルゴリズム

3) PCAST 文の実装

PCAST 文の実装の場合には, 各プロセッサが所有する配列要素を通信バッファにパッキングする部分は PASS 文と同様であるが, PCAST では全てのプロセッサにそれをブロードキャストする. したがって, PCAST 文のアルゴリズムは図 2.24のようにまとめられる. MPI を用いた実装では, **broadcast** に関数`MPI_Bcast` を用いる.

```
procedure PCAST(source,dest)
    pack [buff] ← { source(1,1),source(2,1),...,source(1,2),source(2,2),....}
    for i=1..P
      broadcast [buff] root_processor(i)
      unpack [buff] → { dest(i),dest(i+P),....}
    endfor
end procedure
```

図 2.24: メッセージパッシングモデルでの PCAST 文アルゴリズム

4) BCAST 文の実装

BCAST 文は, 文字通りブロードキャストで実現できる. MPI では`MPI_Bcast` がこれに相当する.

2.5.3 分散メモリの特徴を取り込む拡張

MPI を用いた処理系を実装するにあたり, 分散メモリでの利用を強く意識した 3 つの構文と新たなデータ構造を ADETRAN 4 に取り込んだ.

1) unify 文

分散メモリアーキテクチャではセグメントベクトルを常にグローバルな変数または配列とすることはできない．例えば，PDO ブロック内でプロセッサが担当する配列要素を用いて計算した結果をセグメントベクトルに格納した場合，セグメントベクトルはプロセッサ毎に値が異なるローカル変数となる．

利用者がセグメントベクトルを意識的にローカルに利用することは問題がないが，ADETRAN4 の G 構文や P 構文の引数にはグローバル変数を用いなければいけない制約がある．また，特定プロセッサで計算された値をセグメントベクトルの共通セットとしての値に代表させたいことが生じる．その様な分散メモリアーキテクチャ上で共通セットのコンシステンシを制御する命令として UNIFY 文を設けた．UNIFY 文はメッセージパッシングモデルの立場では，明示的な broadcast に相当する．

UNIFY 文の例を次に示す．1 行目は on 節によって指定された i 番目の論理プロセッサ上のセグメントベクトル f でもって，そのシステム上の f の値を統一させるもの．2 行目は セグメントベクトルの範囲を指定することにより，指定された配列要素のみを統一させるものである．

```
unify(f) on(i)
unify(v(1:k)) on(j)
```

MPI を用いた実装では，文字通り関数 MPI_Bcast を用いる．

2) preduce 文

PREDUCE 文は個々のプロセッサで異なる値を保持するときに，プロセッサ間にまたがって和，積，最大値，最小値等の演算を行うものである．一般にリダクションもしくはベクトルリダクションと呼ばれる次の演算を分散メモリアーキテクチャで実現するものである．

```
s=0.0
do i=1,n
   s=s+a(i)+x(i)
enddo
```

PREDUCE 文の例を次に示す．1 行目は，個々の物理プロセッサ上のセグメントベクトル f についてプロセッサ間での総和をとる操作である．2 行目は，セグメントベクトル v の 1 から n 番成分に関して全てのプロセッサ間での総和をとる操作である．これらの操作では指定したセグメントベクトルに対して結果が格納される．3 行目は，v の最大値をセグメントベクトル vmax に格納するものである．

```
preduce(sum,f)
preduce(sum,v(1:n))
preduce(max,vmax,v)
```

MPI を用いた実装では，文字通り関数 MPI_Allreduce を用いる．

3) on 節

UNIFY 文の説明部分で現われた on 節について, 別の構文における使用法を示す. ADETRAN4 では, G 構文において全プロセッサの動作の coherence を保つためグローバル変数の指定を必要とする. on 節は指定した論理プロセッサ上の値でもって全てのプロセッサの制御を行う場合に使用したり, 明示的な同期指示にも利用可能である.

先に示した用例の他に, on 節は GIF 文で利用される.

```
unify(f) on(i)
gif(k.gt.1) on(j) then
   ....
gendif
```

4) 縮退セグメント配列

v(1,/n/) もしくはv(/n/,1) という形のセグメント配列は, 特に行列問題によくあらわれる. 行列の次数が巨大になったときに, ベクトルそのものを分割して保持する必要が生じるためである. この '(1,' という明示的なインデックスの表記を常に行うことは冗長であるとともに, 本来方向属性を持たないベクトル (行ベクトル, 列ベクトルの表記はあるが) に対して PDO 文の方向属性の制限によって使用が困難な部分が生じてしまう.

そこで, 明示的なインデックスを省略できる縮退セグメント配列を定義している. 表記法は

v(/n/) または 拡張次元のついた v(/n/)(m)

実装上は '(1,' の付いたセグメント配列と変わらないが, 方向属性を有しないため接頭語 (タグ) がl$ の形になる (33頁ならびに図 2.20を参考にされたい).

2.5.4 他言語とのインターフェイス

MPI を利用したメッセージパッシングコードに変換する上で, 他言語との相互乗り入れを考慮した実装を行っている. ADETRAN4 には G サブプログラムと L サブプログラムの 2 階層が存在している. G サブプログラムは全てのセグメント配列を, X 方向属性になるように受け渡しする. L サブプログラムは PURE なサブプログラムとして扱うことで, 他言語との相互利用が可能である. 他言語との相互呼び出しの例を図 2.25に示す.

また, 通信ライブラリ MPI とのインタフェイスとして次の関数を定義している.

- ADE4_Comm_world(): ADETRAN4 処理系が規定する MPI プロセスグループ内コミュニケータ. 他言語から ADETRAN4 を呼び出す場合には利用するプロセスグループのコミュニケータを指定する.
- ADE4_Noproc(): ADETRAN4 処理系に割り当てられる実行時のプロセッサ数を返す. MPI_Comm_size(ADE4_Comm_world(),...) と同様の機能を持つ.

- `ADE4_Idproc()`: 実行時に物理プロセッサに割り付けられる ID(プロセッサ番号) を返す. `MPI_Comm_rank(ADE4_Comm_world(),...)` の戻り値に 1 足したもの.
- `ADE4_Owner(x)`: 論理プロセッサ x が割り当てられる物理プロセッサの ID を返す.
- `ADE4_Phase(x)`: PDO 文内で処理されるインデックス x における, 処理範囲 (多重処理フェイズ) を返す.

ADETRAN4 プログラムの中から MPI 通信関数を呼び出すことができる. 注意すべきこととして, PDO 文の範囲指定が実プロセッサ数と異なる場合には通信デッドロックを起こす可能性がある点, また ADETRAN4 では PASS, PCAST, UNIFY 文等のデータの編集操作以外ではプロセッサ間の同期を行わないので, 利用者の責任で適時`MPI_Barrier` を呼び出す必要がある点等である. また,ADETRAN4 のプロセッサ ID は 1 から始まることも注意が必要である.

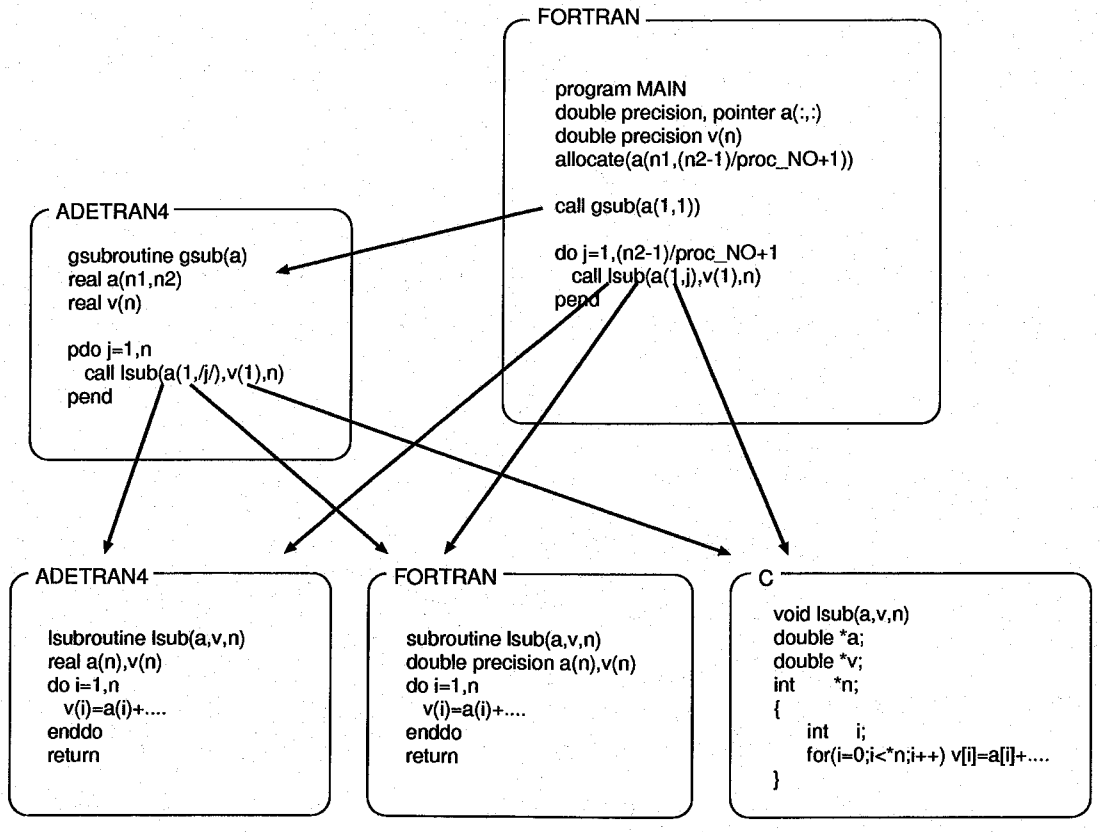


図 2.25: ADETRAN4 と他言語との相互呼び出し, SPMD 形式の Fortran プログラム MAIN から ADETRAN4 の G サブプログラム gsub を call する例 (左中段) と, ADETRAN4/FORTRAN/C のサブプログラム lsub を call する例 (下段).

```
      call MPI_Barrier(ADE4_Comm_world(),ierr)
      pdo i=1,ADE4_Noproc()
!
!  send data to next processor(myrank+1)
!
         isend_rank=mod((i+1)-1,ADE4_Noproc())+1
         irecv_rank=mod((i-1)-1,ADE4_Noproc())+1
         call MPI_Send(data,1,MPI_INTEGER, \
                       isend_rank-1,tag,ADE4_Comm_world(),ierr)
         call MPI_Recv(data,1,MPI_INTEGER, \
                       irecv_rank-1,tag,ADE4_Comm_world(),status,ierr)
      pend
      call MPI_Barrier(ADE4_Comm_world(),ierr)
```

図 2.26: ADETRAN4 から MPI の呼び出し

まとめ

本章のまとめとして, 交互方向操作に基づく並列処理モデル ADEPS は, 工学諸問題に現れる様々な計算スキームを自然な形の 1 次元化もしくは分解を行い, 異なる方向からの単純化された 1 次元問題の扱いを行うことでアルゴリズムを構成しようとするモデルである.

並列処理モデル ADEPS を実践するための, 並列言語処理系 ADETRAN4 を開発し ADENART, VPP500 での利用を経て, 現在, MPI+Fortran90 が利用可能な並列計算機に対しての利用可能な状態になっている. ここで 開発した ADETRAN4 処理系は次章以下に述べる諸問題への適用に利用されている. 開発した処理系の適応可能プラットフォームが極めて広範囲に渡るため, 同一のソースファイルを用いて各種プラットフォームでの利用が可能となっている.

第 3 章

行列問題に対するプログラム実装と計算量モデル

前章では, 並列処理様式 ADEPS を表現し各種並列計算機上で実行するための処理系開発について示した. 本章以降では, 開発した処理系を利用した幾つかの基本的な問題への適用例に付いて報告する. 本 4 章では, 行列計算の典型例から LU 分解, ハウスホルダー 3 重対角化, divide and conquer 法についてその適用結果を示す. LU 分解では, ブロックサイクリック分割を最適化するための評価式の導出を行い. ハウスホルダー 3 重対角化では 行列データの対称性を活かした効率的なアルゴリズムの提案を行い, コストモデルの導出によりその効果を示している. 最後の divide and conquer 法では, 既に知られている手法を拡張したアルゴリズムを提案し並列計算機での実験によりその優位性を示している.

3.1 縦ブロック分割並列 LU 分解

論文 [47] では密行列計算での代表的な計算項目である線形方程式を解くための LU 分解の並列処理について扱った. LU 分解は線形方程式

$$Ax = b \tag{3.1.1}$$

を求解する際に, $A = LU$ の形に一旦分解し

$$Ly = b \tag{3.1.2}$$

$$Ux = y \tag{3.1.3}$$

の 2 つの方程式を解くことに帰着するものである. ここで, L および U はそれぞれ, 下三角行列ならびに上三角行列である. L, U は三角行列であるため上記方程式は簡単な代入操作によって解くことが可能である.

LU 分解 (特に密行列に限った) の並列化は数多くのアルゴリズムが提案され [11, 16, 52, 56], 数値計算ライブラリ [2, 8, 12] で実装されている. その大勢は行列をブロック行列に分解した, ブロック LU 分解やタイリング技術を併用するものが多い. しかし, それら手法はキャッシュを意識した実装方法でありベクトルプロセッサを意識した手法ではない.

以下本節では, ベクトルプロセッサでの実装を意識した縦ブロック分割 LU 分解法の並列化を提案するとともにそのブロックサイズと利用するプロセッサ数の関係についてモデル化し議論する.

3.1.1 ブロック LU 分解

A を次の様に書き直す,

$$A = \begin{pmatrix} L_{11} \\ L_{21} \\ \vdots \\ L_{M1} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & \cdots & U_{1M} \end{pmatrix} + \begin{pmatrix} 0 \\ L_{22} \\ \vdots \\ L_{M2} \end{pmatrix} \begin{pmatrix} 0 & U_{22} & \cdots & U_{2M} \end{pmatrix} + \cdots \tag{3.1.4}$$

今, 第一項目の作り方から A の上 1 列ブロック, 左 1 行ブロック分は L_{*1}, U_{1*} の寄与のみからなるので次のような方程式が成り立つ.

$$L_{11}U_{11} = A_{11} \tag{3.1.5}$$

$$L_{21}U_{11} = A_{21},\ L_{31}U_{11} = A_{31},\ \cdots,\ L_{M1}U_{11} = A_{M1} \tag{3.1.6}$$

$$L_{11}U_{12} = A_{12},\ L_{11}U_{13} = A_{13},\ \cdots,\ L_{11}U_{1M} = A_{1M} \tag{3.1.7}$$

式 (3.1.5) は通常の LU 分解である. L_{11}, U_{11} が三角行列であることから, それ以外の L_{*1}, U_{1*} は代入操作により容易に計算される. 以下同様にして再帰的に同じ手続きを行うことでアルゴリズムが完結する. これがブロック化 LU 分解のアルゴリズムであり, アルゴリズム全体の手続きは次に示すものとなる.

```
1:do k=1, M                          6:   do i=k+1, M
2:   A_kk → L_kk U_kk                7:      do j=k+1, M
3:   do i=k+1, M                     8:         A_ij := A_ij − A_ik A_kj
4:      A_ik:=A_ik U_kk^-1, A_ki:=L_kk^-1 A_ki   9:      enddo
5:   enddo                           10:  enddo
                                     11:enddo
```

図 3.1: 正方ブロックを単位とするブロック LU 分解

3.1.2 縦ブロック化アルゴリズム

先に述べたブロックアルゴリズムは, キャッシュを意識したものであったためにブロックの一辺が十分な大きさではなくベクトル長がベクトルプロセッサで扱うには不十分なものと言える.

そこで, 式 (3.1.4) の第一項の縦のブロックを一括して扱うことによって, 縦ブロック化アルゴリズムが考案できる. 本手法は 2 つの限定されたブロックのみを用いて計算できる点で, メモリの局所性と計算単位 (粒度) を大きくできる. そのため, 分散メモリ型並列計算機上で効率的な手法

といえる．また先に述べた通り，縦ブロック化によってベクトルの最大長を保証しているのでベクトルプロセッサ上で効果的である．以下本アルゴリズムの分散メモリ型並列計算機上での実装について示す．

行列 A を $A = \left[\mathcal{A}_1 \middle| \mathcal{A}_2 \middle| \cdots \middle| \mathcal{A}_M \right]$ と縦ブロックに分割する．このブロックを分散メモリ上の配列に格納する方法として，プロセッサ番号に対応してラウンドロビン方式で割り当てる幅付サイクリックまたはブロックサイクリックと呼ばれる方式を用いる．分割されたデータは個々のローカルメモリ上で，例えばプロセッサ番号 1 のローカルメモリ上には $\{\mathcal{A}_1, \mathcal{A}_{P+1}, \ldots\}$ の順にデータが並ぶように配列を確保し行列データを格納する．実際には `A(N, m, M/P)` の形で配列を確保し縦ブロックを格納する．ここで P はプロセッサ数，N, m はそれぞれ行列サイズとブロック幅である．

つまり，縦ブロックアルゴリズムのブロックとメモリ分割方式のブロックを同一視し，論理的なブロックと物理的なブロックを一致させる様なマッピングを行う．本割当て方式ではブロックが他のプロセッサにまたがって配置されず，ブロック内処理がデータ転送のために途中中断することがない．

先行して計算される基本軸ブロックは単一のプロセッサのメモリ上に存在し，それを保有するプロセッサのみが基本軸を計算可能である．続いて行なうガウス消去で，先行計算された基本軸が他のプロセッサで必要となるが，そのときには基本軸を必要としているプロセッサに転送する．LU 分解では全プロセッサが基本軸を必要とするので 1 対全通信のブロードキャストを行う．

分解列のブロードキャストも含めて図 3.2 に示すステップを踏むことで LU 分解がなされる．k_b や j_b など下添字に b をつけたものはブロックについての繰り返しを制御する変数である．ブロック $\mathcal{A}_i$ につく下括弧の添字は行列全体ではなくブロック内での要素の位置を表す．$\mathcal{W}_*$ は各プロセッサのローカルメモリに分解軸ブロックのコピーを作っておき同一値を保持する一時記憶領域である．

以下本論文では本手法を縦ブロック分割並列 LU 分解 (VBPLU) と呼ぶ．

1:**do** k_b=1, M

2: **if**(owner of $\mathcal{A}_{k_b}$)**then**

3: **do** k_0=1, m; $k=(k_b-1)m+k_0$

4: ipvt(k)←maxloc{abs($\mathcal{A}_{k_b(i,k_0)}$)} $(i=k+1...N)$

5: $\mathcal{A}_{k_b(\mathrm{ipvt}(k),k_0)} \leftrightarrow \mathcal{A}_{k_b(k,k_0)}$

6: $\mathcal{A}_{k_b(i,k_0)} \leftarrow \mathcal{A}_{k_b(i,k_0)} / \mathcal{A}_{k_b(k,k_0)}$, $(i=k+1...N)$

7: **do** j=1, $m-k_0$

8: $\mathcal{A}_{k_b(\mathrm{ipvt}(k),k_0+j)} \leftrightarrow \mathcal{A}_{k_b(k,k_0+j)}$

9: $\mathcal{A}_{k_b(i,k_0+j)} \leftarrow \mathcal{A}_{k_b(i,k_0+j)} - \mathcal{A}_{k_b(i,k_0)} \mathcal{A}_{k_b(k,k_0+j)}$, $(i=k+1...N)$

10: **enddo**

11: **enddo**

12: **endif**

13: **if**($k_b == M$) **goto** 32:

14: **broadcast** $\mathcal{A}_{k_b(i,j)} \rightarrow \mathcal{W}_{*(i,j)}$, root is owner of $\mathcal{A}_{k_b}$ $(i=(k_b-1)m+1...N,\ j=1...m)$

15: **broadcast** ipvt(k)→ipvt(k), root is owner of $\mathcal{A}_{k_b}$ $(k=(k_b-1)m+1...k_bm)$

16: **do** k_0=1, m; k=$(k_b-1)m+k_0$

17: $\mathcal{W}_{*(\mathrm{ipvt}(k),j)} \leftrightarrow \mathcal{W}_{*(k,j)}$, $(j=1...k_0-1)$

18: **parallel do** j_b=k_b+1, M

19: **if**(owner of $\mathcal{A}_{j_b}$) $\mathcal{A}_{j_b(\mathrm{ipvt}(k),j)} \leftrightarrow \mathcal{A}_{j_b(k,j)}$, $(j=1...m)$

20: **enddo**

21: **enddo**

22: **parallel do** j_b=k_b+1, M

23: **if**(owner of $\mathcal{A}_{j_b}$)**then**

24: **do** k_0=1, m; $k=(k_b-1)m+k_0$

25: **do** j=1, m

26: $\mathcal{A}_{j_b(i,j)} \leftarrow \mathcal{A}_{j_b(i,j)} - \mathcal{W}_{*(i,k_0)} \mathcal{A}_{j_b(k,j)}$, $(i=k+1...N)$

27: **enddo**

28: **enddo**

29: **endif**

30: **enddo**

31:**enddo**

32:**end of algorithm**

図 3.2: 縦ブロック分割並列 LU 分解 (VBPLU 法)

3.1.3 縦ブロック分割 LU 分解法のモデル化

ブロックサイクリック分割に基づく VBPLU 法はブロック幅 (m) の設定により様々な分割を含むことができる．例えば $m = 1$ とすればサイクリック分割であり，$m = N/P$ とすればブロック分割である．極端な場合として $m = N$ では逐次実行を意味する．ブロック分割による LU 分解はアルゴリズムの後半にプロセッサが遊ぶ状態を招くため性能が出ないことは明らかである．またサイクリック分割ではデータ転送の回数が多く性能を劣化させる．ブロックとサイクリックの中間であるブロックサイクリックはブロック幅の設定により両者の欠点を緩和し，両者の方法よりも計算時間を改善し最良にするポイントが存在することが予想される．本節では VBPLU 法の実行ステップについてのモデル化を行い，次節において実行時間を最良にするブロック幅についての知見を与える．

モデル化の前提として使用する計算機はプロセッサ数 P 台の分散メモリ型の並列計算機で行列 (サイズ:N) は列に対して幅 m のブロックサイクリック分割をする．本論文で考察する縦ブロック分割 LU 分解の実行タイムチャートは図 3.3に示す様になる．初めに分解列を保持するプロセッサが分解列内のピボッティングと対角成分によるスケーリング (図 3.2の 4~6 行に対応) を行うとともにガウス消去 (7~10 行) を行う．次にピボッティング情報と分解列をブロードキャストで全プロセッサに放送し (14, 15 行)，同期後，各プロセッサが一斉にガウス消去を始める (23~31 行)．ガウス消去は個々のプロセッサが保持する未消去ブロックを順に処理していき，ステップの最後に再度同期をとり次のステップに進むものとする．

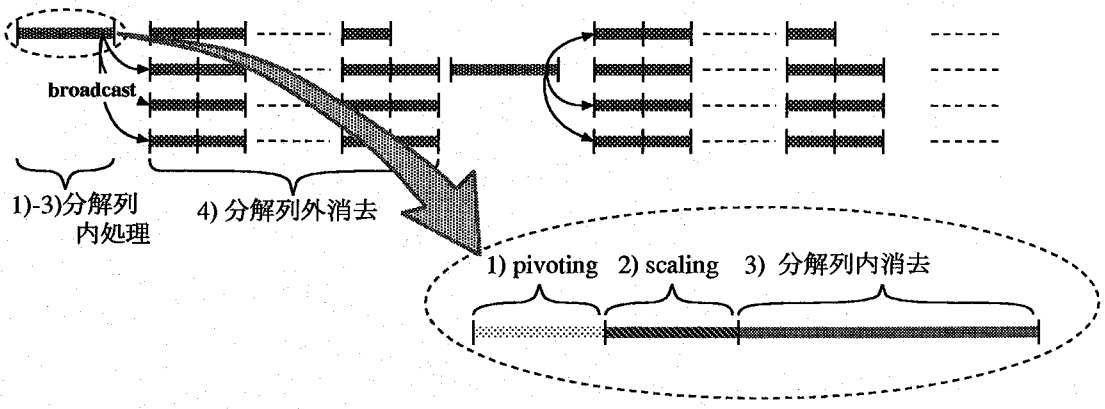


図 3.3: LU 分解の実行タイムチャート

計算部分の評価

1), 2) ピボッティング並びにスケーリングは逐次型での計算量と一致する．行の交換を全て行なうとすると，分解列内での対角ブロック部分，分解列外での非対角ブロック部分で分けることができる．浮動小数の比較演算と交換，乗算に要する単位時間をそれぞれ τ_{cmp}, $\tau_{\leftrightarrow}$, τ_* とおくと，ピボッテイング + 行交換とスケーリングに要す時間 T_{pivot}, T_{scale} は以下の様になる．

$$T_{\mathrm{pivot}} = \tau_{\mathrm{cmp}} \sum_{i=1}^{N-1} i + \tau_{\leftrightarrow} \left((M-1) \sum_{i=2}^{m} i + \sum_{i=2}^{m^*} i + (M-1)m^2 \right) \tag{3.1.8}$$

$$= \tau_{\mathrm{cmp}} \frac{N(N-1)}{2} + \tau_{\leftrightarrow} \frac{m(3m+1)(M-1) + m^*(m^*+1)}{2} \tag{3.1.9}$$

$$T_{\mathrm{scale}} = \tau_* \frac{N(N-1)}{2} \tag{3.1.10}$$

ここで $m^* = N - (M-1)m$ で最終端ブロックの幅を表し, $M = \lceil N/m \rceil$ でありブロックの個数を表す. なお記号 $\lceil \cdot \rceil$ は小数点以下切上げを意味する.

3) 分解ブロック内のガウス消去はステップ i における幅 m, 長さ $N - mi$ の矩形領域でのガウス消去に必要な積和計算を i について総和をとることで見積ることができる. 積和計算の単位処理時間を $\tau_{(*,+)}$ とおくと, この部分の計算時間 $T_{\mathrm{gauss\_pvt}}$ は次式となる.

$$T_{\mathrm{gauss\_pvt}} = \tau_{(*,+)} \left(\sum_{k_b=1}^{M-1} \sum_{i=1}^{m-1} (m-i)(N-(k_b-1)m-i) + \sum_{i=1}^{m^*-1} (m^*-i)(m^*-i) \right) \tag{3.1.11}$$

$$= \tau_{(*,+)} \left(\frac{(M-1)(m-1)m(6N-3mM+4m-2)}{12} + \frac{(m^*-1)m^*(2m^*-1)}{6} \right) \tag{3.1.12}$$

4) 分解ブロック以外のガウス消去はブロック内の演算量に一つのプロセッサが担当するブロック数 (以下, 多重度と呼ぶ) を乗じたものが対応する. 通常最終端ブロックが幅 m でないので, 最後の実行ブロックが最終端ブロックのみとなるか場合わけが必要となる. したがってこの部分の計算時間 $T_{\mathrm{gauss\_npvt}}$ は次式となる.

$$T_{\mathrm{gauss\_npvt}} = \tau_{(*,+)} \sum_{k_b=1}^{M-1} K(k_b) \tag{3.1.13}$$

$$K(k_b) = \begin{cases} K_1(k_b) & \text{if } M - k_b - 1 \equiv 0 \pmod{P} \quad \text{(3.1.14a)} \\ K_2(k_b) & \text{otherwise} \quad \text{(3.1.14b)} \end{cases}$$

$$K_1(k_b) = (m\gamma_{k_b} + m^*) \sum_{j=1}^{m} (N - (k_b-1)m - j) \tag{3.1.15}$$

$$K_2(k_b) = (m\gamma_{k_b} + m) \sum_{j=1}^{m} (N - (k_b-1)m - j) \tag{3.1.16}$$

ここで $\gamma_{k_b} = \lceil (M-k_b)/P \rceil - 1$ である. $\sum_{k_b} K(k_b)$ を厳密に評価するために $M = \gamma P + \epsilon$, $k_b = \alpha P + \beta$ と分解する. ここで $\gamma = \gamma_0 = \lceil M/P \rceil - 1$, $\epsilon \in [1:P]$ であり, $\gamma+1$ が各プロセッサの持つデータ多重度を表す. また k_b の分解については $0 \le \alpha, \beta \in [1:P]$ とする.

$$M - k_b - 1 = \gamma P + \epsilon - (\alpha P + \beta) - 1 \equiv \epsilon - \beta - 1 \pmod{P} \tag{3.1.17}$$

$$\gamma_{k_b} = \begin{cases} \gamma - \alpha & \text{if } 1 \le \beta < \epsilon \quad \text{(3.1.18a)} \\ \gamma - \alpha - 1 & \text{if } \epsilon \le \beta \le P \quad \text{(3.1.18b)} \end{cases}$$

により $M-k_b-1\equiv 0 \pmod P$ は $\beta=\epsilon-1$ の時 (または $\epsilon=1$ では $\beta=P$ の時) 成立する.

$$\sum_{k_b=1}^{M-1} K(k_b) = \sum_{k_b=1}^{M-1} K_2(k_b) + \sum_{k_b\in\{i|i=\alpha P+\epsilon-1, 1\le i\le M-1\}} (K_1(k_b)-K_2(k_b)) \tag{3.1.19}$$

$$= \sum_{\alpha=0}^{\gamma}\sum_{\beta=1}^{\epsilon-1} K_2(k_b)\Big|_{\substack{k_b=\alpha P+\beta\\ \gamma_{k_b}=\gamma-\alpha}} + \sum_{\alpha=0}^{\gamma}\sum_{\beta=\epsilon}^{P} K_2(k_b)\Big|_{\substack{k_b=\alpha P+\beta\\ \gamma_{k_b}=\gamma-\alpha-1}} \tag{3.1.20}$$

$$\sum_{\alpha=\delta_1(\epsilon)}^{\gamma} (K_1(k_b)-K_2(k_b))\Big|_{k_b=\alpha P+\epsilon-1}$$

$$= \frac{1}{12}(M-\gamma P-1)m^2(\gamma+1)(\gamma+2)(6N-3Mm+m\gamma P+3m-3) \tag{3.1.21}$$

$$+ \frac{1}{12}(P-M+\gamma P+1)m^2\gamma(\gamma+1)(6N-3Mm+m\gamma P+3m-3-Pm)$$

$$+ \Big(N-Mm+\frac{1}{2}(m(\gamma-\delta_1(\epsilon))P+3m-1)\Big)m(m^*-m)(\gamma-\delta_1(\epsilon)+1)$$

通信部分の評価

ブロック分割アルゴリズムに現れるブロードキャスト通信は通信の立上り時間 (τ_{startup}) とデータの転送速度 ($\tau_{\text{trans\_rate}}$), データ量 (Nm), さらにブロードキャストするプロセッサ数 (P) に依存している. 通信経路はネットワーク形状に依存するが, ここでは高速ネットワークを用いて任意のプロセッサへ高々 $\log P$ 回で到達できるとする. 1 回のブロードキャストあたり $\log P$ 回の通信を発行するから 1 回の通信時間 τ_{broad} は次の様に求められる.

$$\tau_{\text{broad}} = (\tau_{\text{broad\_startup}} + \tau_{\text{trans\_rate}} Nm)\lceil \log_2(2P-1)\rceil \tag{3.1.22}$$

さらにルーティングでの立ち上がり時間が初回の命令発行時のそれより緩和されるとすれば

$$\tau_{\text{broad}} = \tau_{\text{broad\_startup0}} + (\tau_{\text{broad\_startup1}} + \tau_{\text{trans\_rate}} Nm)\lceil \log_2(2P-1)\rceil \tag{3.1.23}$$

ここで $\tau_{\text{broad\_startup}} > \tau_{\text{broad\_startup1}}$. 本モデルの場合ブロードキャストは $M-1$ 回実行することになるのでブロードキャスト全体で要する時間 T_{broad} は次式となる.

$$T_{\text{broad}} = \Big(\tau_{\text{broad\_startup0}} + (\tau_{\text{broad\_startup1}} + \tau_{\text{trans\_rate}} Nm)\lceil \log_2(2P-1)\rceil\Big)(M-1) \tag{3.1.24}$$

ここで N が十分大きい時, $M-1 = N/m$ とできるので N の最高次の N^2 までで近似すると $T_{\text{broad}} = \tau_{\text{trans\_rate}} N^2 \lceil \log_2(2P-1)\rceil$ となり通信立ち上がりが無視できるとともにプロセッサ数 (P) と行列サイズ (N) に依存しブロック幅 (m) に依存しない. 逆に N が小さい時は通信立ち上がりは無視できないことになる.

その他部分の評価

本手法は核の部分で 4 重ループを構成するので最深と上位ループの回転数のバランスによってオーバヘッドが実行時間に顕著に現れる. 特に最深ループの発行数はベクトルプロセッサでの評

価に重要な $N_{1/2}$ を乗じることでベクトルパイプライン立ち上がり時間の総和として評価できる. ループ先頭に重み $(\alpha_{(\cdot)})$ をかけて数え上げると次式 (3.1.26) のようになる. (最内側で $\alpha_3 = \alpha_7 = \tau_{(*,+)} * N_{1/2}$ とおくとベクトルプロセッサの評価に利用できることに注意.)

$$T_{\text{loop\_head}} = \sum_{k_b=1}^{M}(\alpha_1 + \sum_{k_0=1}^{m}(\alpha_2 + \sum_{j=1}^{m-k_0}\alpha_3)) + \sum_{k_b=1}^{M-1}(\alpha_4 + \sum_{k_0=1}^{m}(\alpha_5 + \sum^{\gamma_k+1}(\alpha_6 + \sum_{j=1}^{m \text{ or } m^*}\alpha_7))) \tag{3.1.25}$$

$$\begin{aligned} &= M\left(\alpha_1 + m\alpha_2 + \frac{1}{2}(m-1)m\alpha_3\right) + (M-1)(\alpha_4 + m\alpha_5) \\ &\quad + \frac{1}{2}(\gamma+1)(2M-2-\gamma P)m(\alpha_6 + m\alpha_7) + m(m^*-m)(\gamma - \delta_1(\epsilon) + 1)\alpha_7 \end{aligned} \tag{3.1.26}$$

ベクトル化コンパイラが多重ループ内のベクトルパイプラインの立ち上がりを計算の裏側に隠蔽する最適化をした場合には, 立ち上がりのオーバヘッドは $\sum\alpha_3 \Rightarrow \alpha_3$, $\sum\alpha_7 \Rightarrow \alpha_7$ と変更でき, $T_{\text{loop\_head}}$ は次式の様になる.

$$\begin{aligned} T^*_{\text{loop\_head}} = M\left(\alpha_1 + m(\alpha_2 + \alpha_3)\right) + (M-1)(\alpha_4 + m\alpha_5) \\ + \frac{1}{2}(\gamma+1)(2M-2-\gamma P)m(\alpha_6 + \alpha_7) \end{aligned} \tag{3.1.27}$$

3.1.4 計算見積り時間 T_{comp} の評価

以上の結果より計算量から要請される計算時間

$$T_{\text{comp}} = T_{\text{pivot}} + T_{\text{scale}} + T_{\text{gauss\_pvt}} + T_{\text{gauss\_npvt}} + T_{\text{broad}} + T_{\text{loop\_head}} \tag{3.1.28}$$

が独立変数 N, P, m と並列計算機を特徴づけるいくつかのパラメータを用いて見積もることができる. 本論文ではグラフをプロットしそこから得られる定性的な振舞いとともに, 問題サイズ, 利用するプロセッサ数に対して最適なブロック幅をそこから読みとることをする. 特に使用するプロセッサエレメントにスカラプロセッサ, ベクトルプロセッサを想定しそれぞれについて評価式を調べていく.

1) スカラプロセッサの場合

スカラプロセッサ, 特に RISC を使用する場合はキャッシュミスの影響を考慮しなくてはならない. しかし様々な不確定 (しかも確率的で外乱も含む) 要素が絡んでくるので正確な評価は不可能に近い, そこで不確定要素が一切入らない理想的な状態を想定する. 条件として 1) 十分にキャッシュサイズがありミスヒットが起こらない, 2) スーパスカラ方式によりレジスタへのロード命令と演算命令が同時処理される, 3) レジスタハザード等のパイプライン性能を劣化させる要因は一切起こらない, これらを仮定する.

単位処理時間をクロックサイクルで正規化し $\tau_{\text{cmp}} = \tau_{\leftrightarrow} = 10, \tau_* = 1, \tau_{(*,+)} = 2$ とおく. ループのオーバヘッドは $\alpha_{i=1,2,4,5,6} = 10, \alpha_{i=3,7} = 20$ とする. 通信に関してはスタートアッ

プを $\tau_{\text{broad\_startup0}} = 10^3$, $\tau_{\text{broad\_startup1}} = 5 * 10^3$, 通信速度を $\tau_{\text{trans\_rate}} = 5$ とする．これはクロック周波数 150MHz のプロセッサと 240MB/sec のネットワークを組合せたスカラ並列システムに相当し，後述の SR2201 のスペックと同等である．このデータを基に，行列サイズ $N = 10, 10^2, 10^3, 10^4$ のそれぞれについて T_{comp} をプロットした (図 3.4)．ブロックの幅は $m = 1$ ... $\max(\min(N, 100), N/10)$, プロセッサ数は $P = 1$ ... $\max(\min(N, 100), N/10)$ の範囲で変化させた．$N = 10$ は並列計算機上で計算するには小さいが，$10^2 \sim 10^4$ は現実的な規模である．メモリに関しても $N = 10^4, P = 8$ で各プロセッサで約 100MByte を使用するが，100MByte 以上のメモリを搭載する機種も多く存在し評価上限として適当である．

2) ベクトルプロセッサの場合

基本的にパラメータはスカラの場合と同様にするが，四則演算により処理速度を正規化するのでスカラに比べて演算処理速度が圧倒的に速くなる．演算については加減・乗算器の同時動作 (チェイニング) を仮定し $\tau_{(*,+)} = 1$ とする．それ以外のパラメータは以下のように修正する．ループの立ち上がりは $\alpha_{i=1,2,4,5,6} = 100, \alpha_3 = \alpha_7 = \tau_{(*,+)} N_{1/2} = 400$ とする．また転送の立ち上がりを $\tau_{\text{broad\_startup0}} = 1.5 * 10^4$, $\tau_{\text{broad\_startup1}} = 0.8 * 10^4$, 転送レートを $\tau_{\text{trans\_rate}} = 20$ とする．これはベクトルパイプラインが 8 段でクロック周波数 125MHz つまりピーク 2.0GFLOPS のベクトルプロセッサと 400MB/sec のネットワークを組み合わせたベクトル並列システムに相当し，後述の VPP300 のスペックに同等である．スカラプロセッサと同様の範囲で，前章で示した多重ループのパイプライン最適化をするものも併せて 2 種類のグラフをプロットした (図 3.5, 3.6).

スカラプロセッサとパイプライン最適化を行なわないベクトルプロセッサのグラフ形状は同様になったが，パイプライン最適化を行なったものはやや形状が異なる結果となった．図 3.4, 3.5, 3.6から読みとれる点は次にまとめられる．

- 各問題サイズごとに，プロセッサ台数に依存しない実行時間を最小にするブロック幅が存在する．
- $N = 10^2, 10^3$ では，通信と演算が均衡しており実行時間を最小にする分割ブロック幅とプロセッサ数の組合せが存在する．つまりプロセッサ台数効果の上限が極数台のレベルで存在している．
- $N = 10^4$ では，現実的なプロセッサ台数で台数効果がよく現われている．
- $N = 10^3$ で見た時，最適なブロック幅はスカラでは $4 \sim 20$, ベクトルでは $10 \sim 50$ である

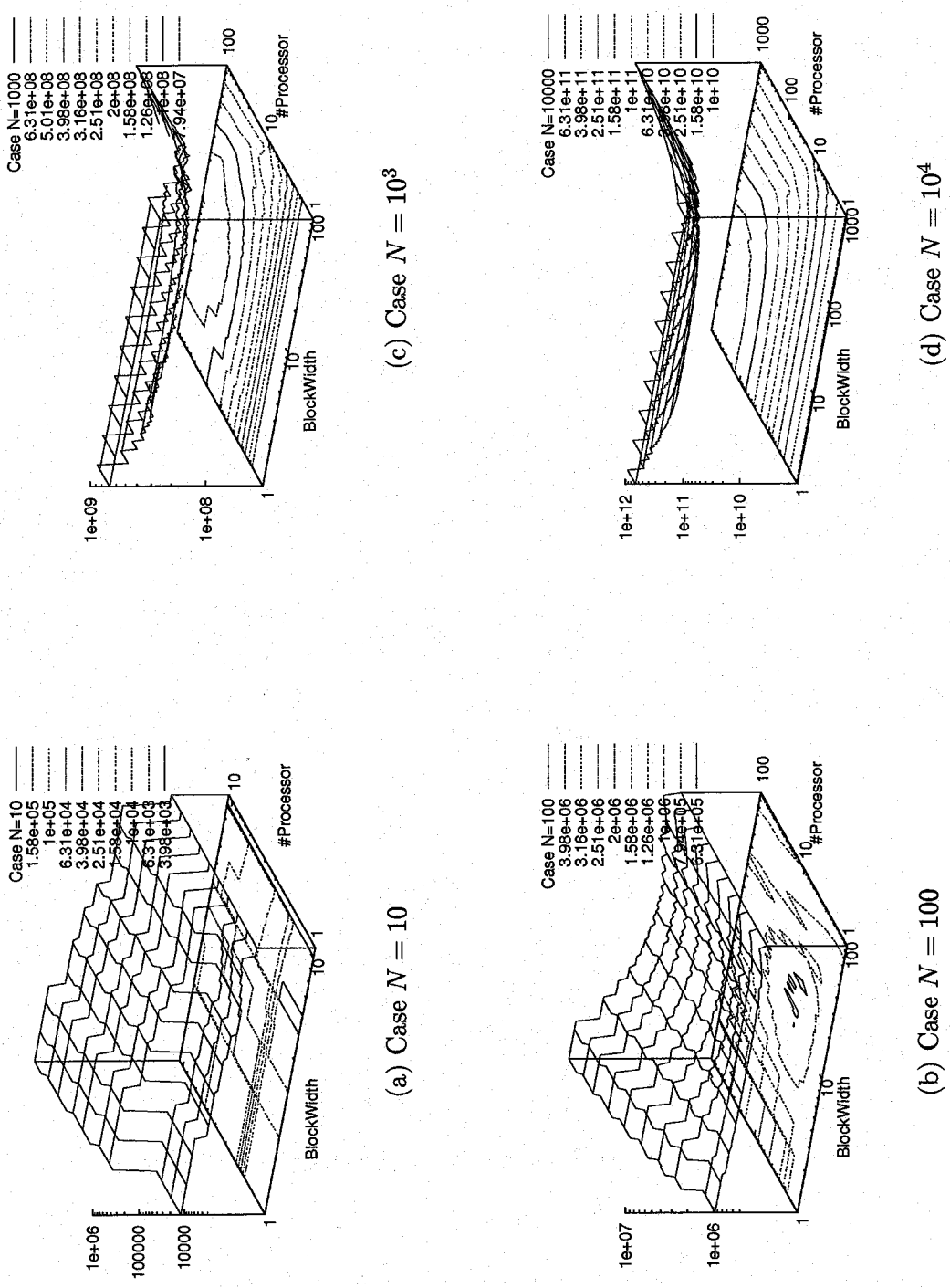
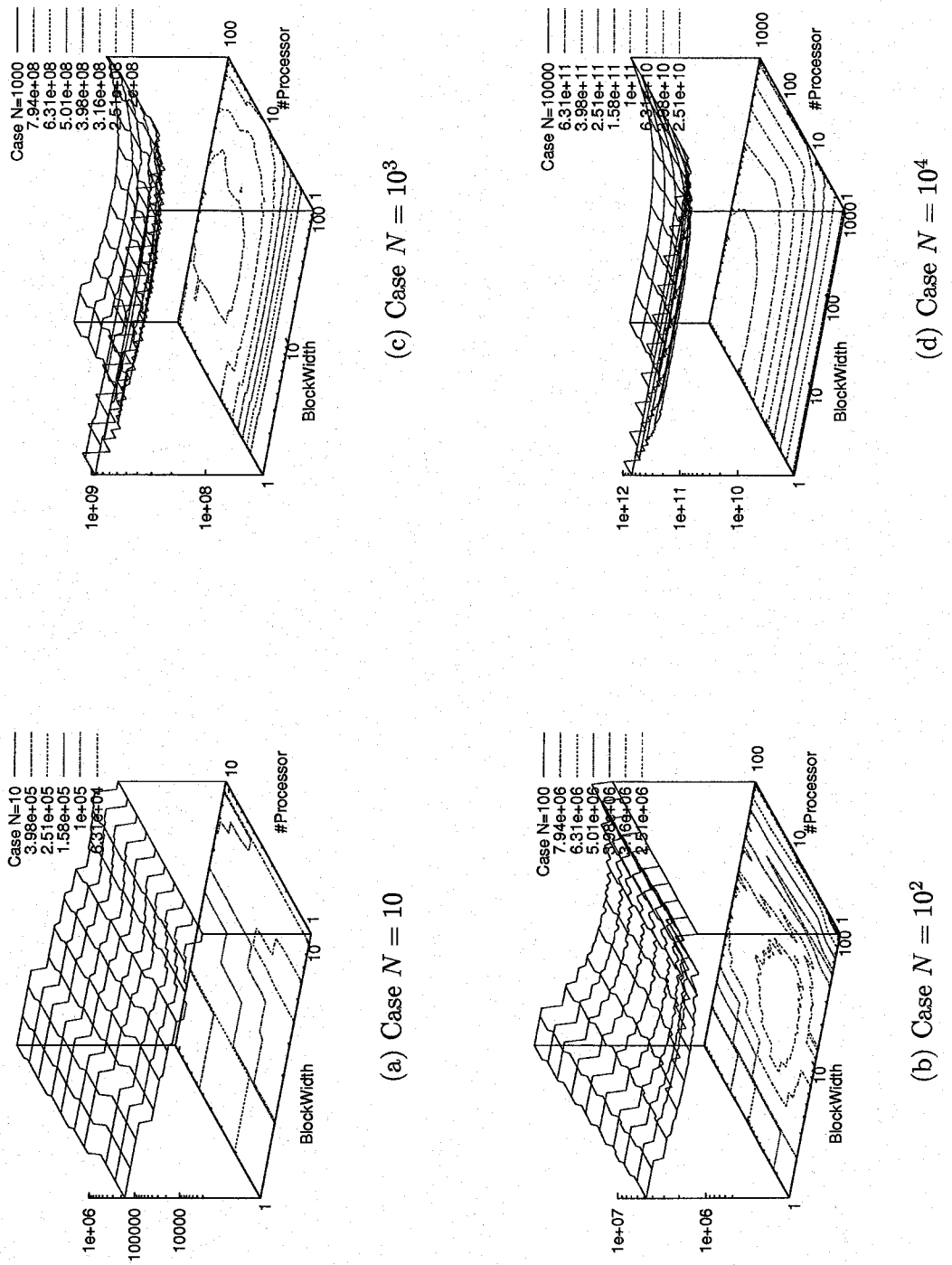


図 3.4: スカラプロセッサを仮定したときの計算量評価 (ブロック幅とプロセッサ数を変化させたものについてコンタープロットした)



(a) Case $N = 10$ (b) Case $N = 10^2$ (c) Case $N = 10^3$ (d) Case $N = 10^4$

図 3.5: ベクトルプロセッサを仮定したときの計算量評価 (ブロック幅とプロセッサ数を変化させたものについてコンタープロットした)

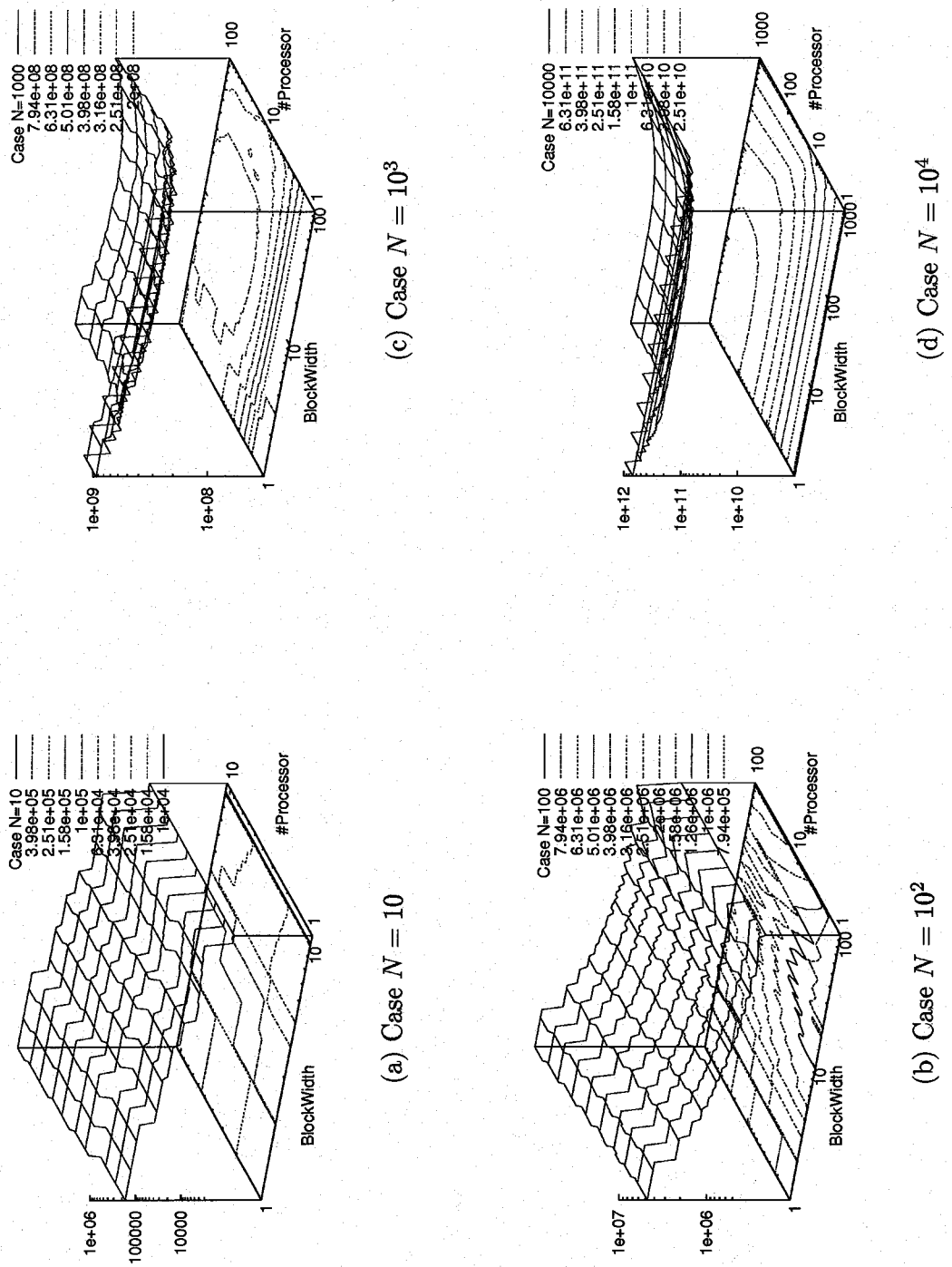


図 3.6: ベクトルプロセッサでパイプライン最適化を仮定したときの計算量評価 (ブロック幅とプロセッサ数を変化させたものについてコンタープロットした)

3.1.5 数値実験によるモデルの検証

SR2201 での実験結果

分散メモリ型スカラ並列計算機として日立 SR2201 で実験を行った. SR2201 は PA-RISC に独自の疑似ベクトル機能を追加したプロセッサと 2 次元ハイパークロスバーの高速ネットワークが特徴である. プロセッサはマシンクロック 150MHz で加減と乗算を同時実行可能でプロセッサ単体のピーク性能は 300MFLOPS である. また疑似ベクトル機能はプリロード機能によるスループットの向上を狙ったものである. さらに浮動小数点演算の他に整数またはロード - ストア命令の同時実行が可能である. プロセッサについてのこれらのスペックは前節で仮定した 3 項目をほぼ満足する. ネットワークの通信性能は 300MB/sec と高速である.

前章で示した, ADETRAN4 を用いて VBPLU ルーチンを作成し, 通信ライブラリに MPI[24] を利用した Fortran90 ソースプログラムに変換し実行した. 作成した ADETRAN4 ソースは約 120 行で, 変換した Fortran90+MPI ソースは約 220 行である.

図 3.7は行列サイズ 10, 100, 500 の 3 ケースについてプロットしたものである. 横軸がブロック幅でグラフ中の括弧内の数字がプロセッサ数であり, 縦軸は実行 (経過) 時間で単位はマイクロ秒である. 図 3.8では行列サイズ 1000 とし, さらに付録で示す多段同時消去を k について 4 段行った場合を併せてプロットした.

$N = 10$ ではプロセッサ数を増加させた時の並列化の効果は全くなく, 逆に $\log P$ で増加する結果となっている. これは通信時間が全体を支配しているためである, 通信時間をプロットしていると見ればブロック幅を増加することで時間が減少することが観測されている. $N = 100$ からはブロック幅が狭い部分では通信時間が大きくなるためプロセッサ数の効果は見えないが, ブロック幅が大きくなるとプロセッサ数効果の演算時間と通信時間を重ね合わせた形でプロットカーブが見え始めている. ブロック幅はプロセッサ数に関係なく $10 \sim 20$ が最適なものといえる. $N = 500$ では演算部分が支配するようになるのでプロセッサ数の効果が完全に現れる. しかしプロセッサ数が 16 を越えると通信部分が支配的になり始めることが読みとれる. また最適なブロック幅は 4 から 20 であり, グラフからは 8 が最適と読める. $N = 1000$ でも $N = 500$ と同様であるが, よりプロセッサ数効果が大きく現れていることが読みとれる. さらに 4 段の同時消去を行った場合には最高で 2 倍の高速化が実現されており大いに効果的であることがわかる. N=1000 では演算部分が支配的になるため, 最適なブロック幅はやや小さくなり 4 から 20 の間に設定できる.

先に示した評価式と測定結果を比較するために, 対応する部分を 2 次元にプロットし直したものを 図 3.9, 3.10に示す. 図 3.7と図 3.9の形状は酷似しており, 図 3.8と図 3.10に関しても同様である. 評価式に含まれるパラメータを変化させたときの振る舞いとして, 通信の立ち上がり時間を大きくするとプロセッサ数増加に伴い実行時間も増加する. またループのオーバヘッドを大きくするとブロック幅を大きくしたときの実行時間増加が起こる. 図 3.11, 図 3.12に $\tau_{\text{broad\_startup0}} = 5 * 10^3, \tau_{\text{broad\_startup1}} = 2 * 10^4, \tau_{\text{trans\_rate}} = 20, \alpha_{i=7} = 400$ とした場合の評価式をプロットした. こちらのグラフの方が実測に近いものが得られている. パラメータ値選定の問題があるが, 評価式は実用の範囲で問題をよく表現していると言えよう.

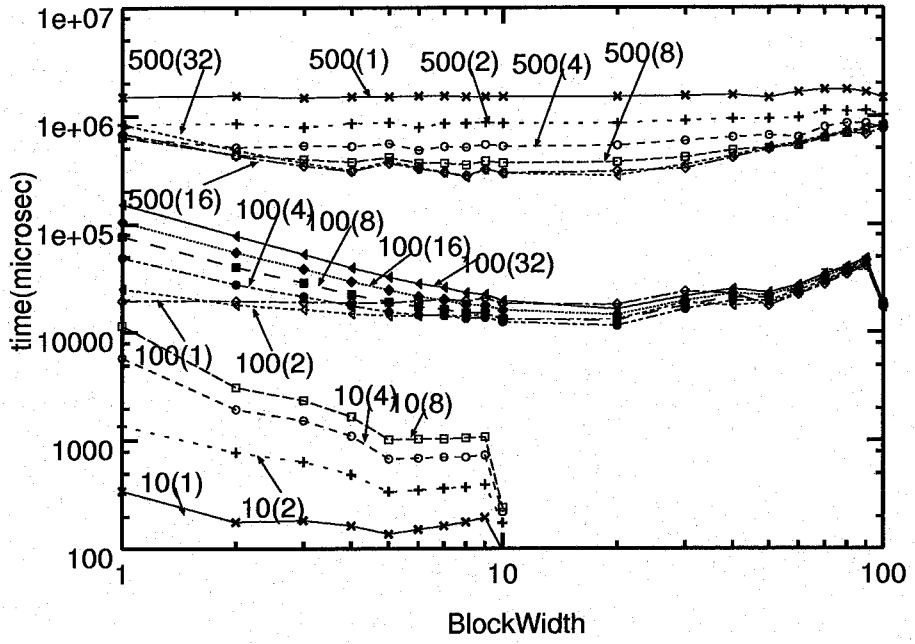


図 3.7: SR2201 での実験結果, 行列サイズ:$N = 10, 100, 500$ に対してプロセッサ台数を $P = 1, 2, 4, 8, 16, 32$ で実行した場合. 図中には'N(P)' の形で示している.

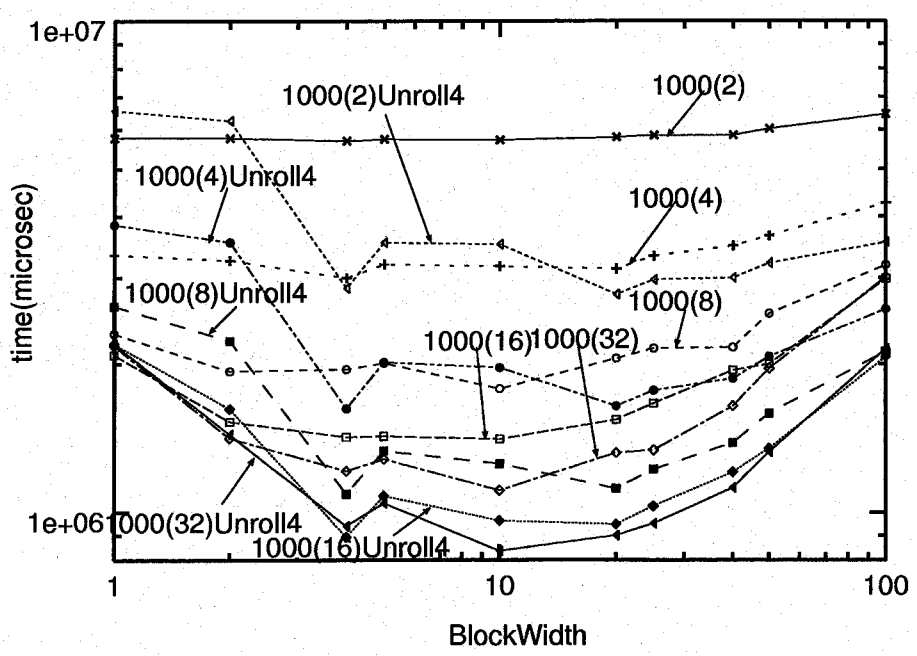


図 3.8: SR2201 での実験結果 (通常版と 4 段アンロール版), 行列サイズ:$N = 1000$ に対してプロセッサ台数を $P = 1, 2, 4, 8, 16, 32$ で実行した場合. 図中には'N(P)' または'N(P)Unroll' の形で示している.

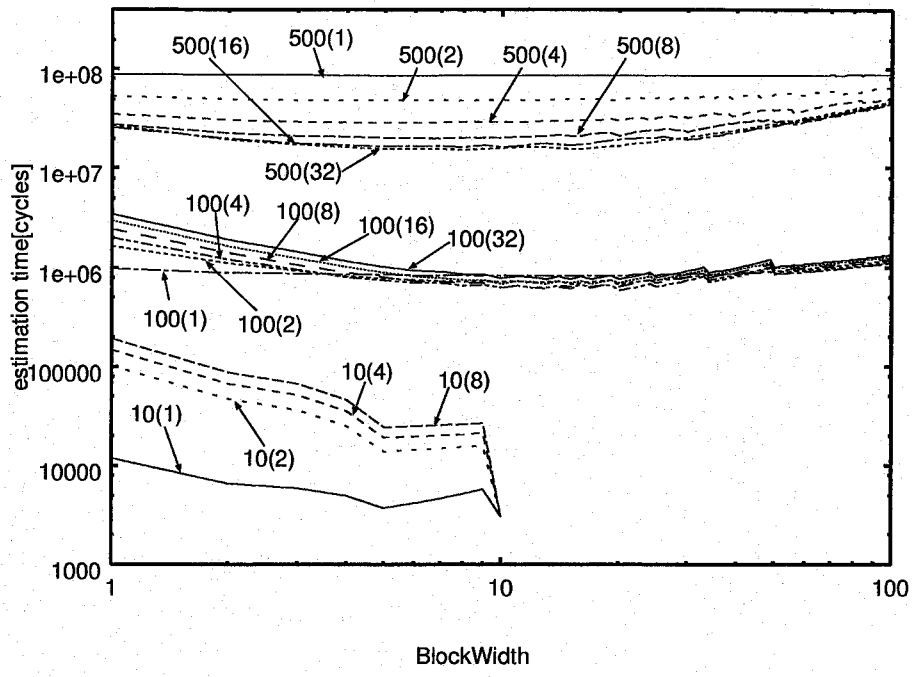


図 3.9: スカラプロセッサでの評価結果, 行列サイズ:$N = 10, 100, 500$ に対して使用するプロセッサ台数を $P = 1, 2, 4, 8, 16, 32$ と仮定した場合. 図中には'N(P)' の形で示している.

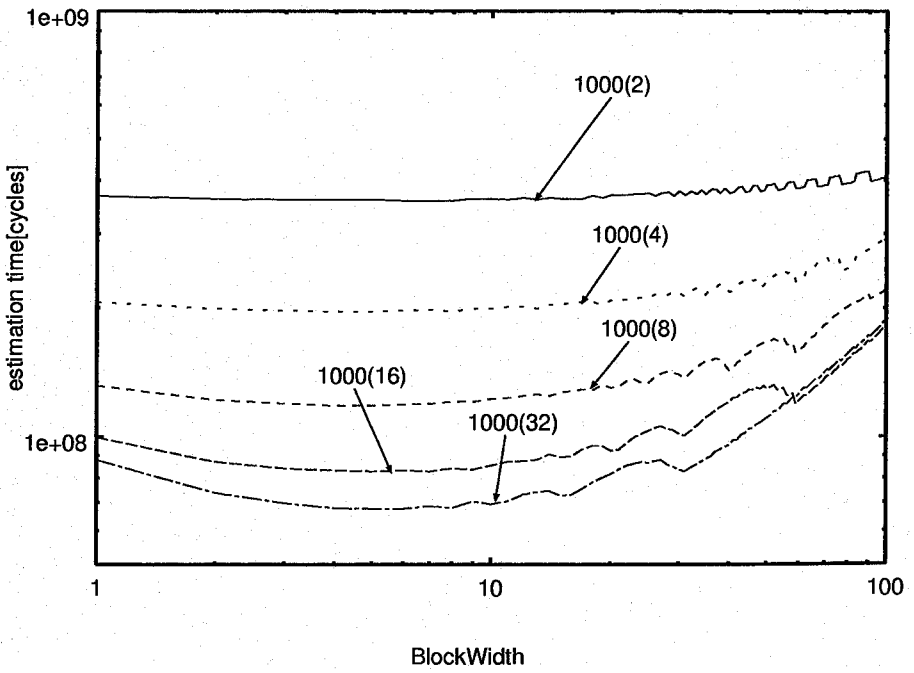


図 3.10: スカラプロセッサでの評価結果, 行列サイズ:$N = 1000$ に対して使用するプロセッサ台数を $P = 1, 2, 4, 8, 16, 32$ と仮定した場合. 図中には'N(P)' の形で示している.

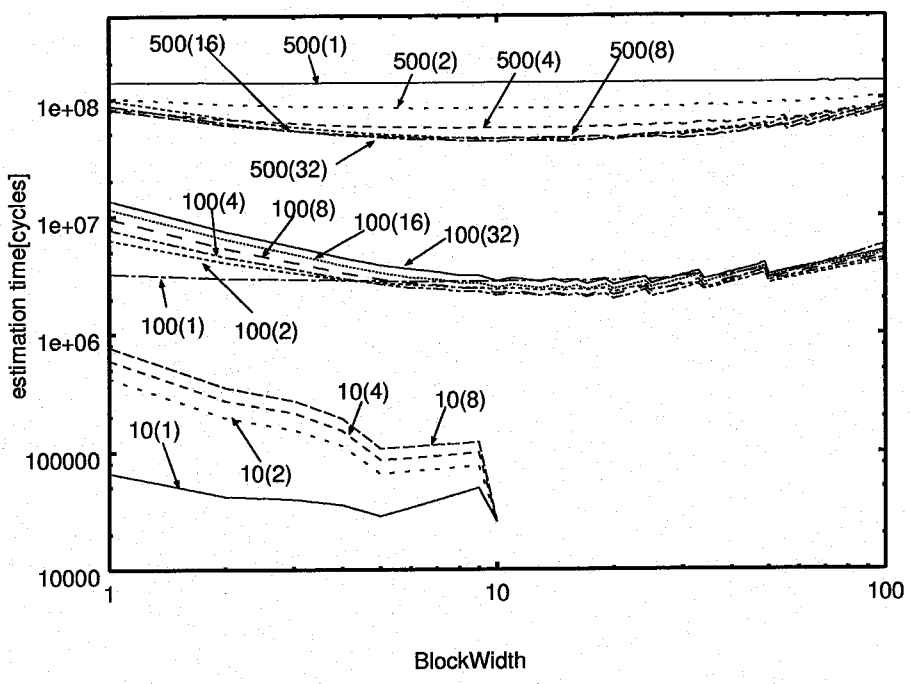


図 3.11: 別のパラメータを用いたスカラプロセッサでの評価結果, 行列サイズ:$N = 10, 100, 500$ に対して使用するプロセッサ台数を $P = 1, 2, 4, 8, 16, 32$ と仮定した場合. 図中には'N(P)' の形で示している.

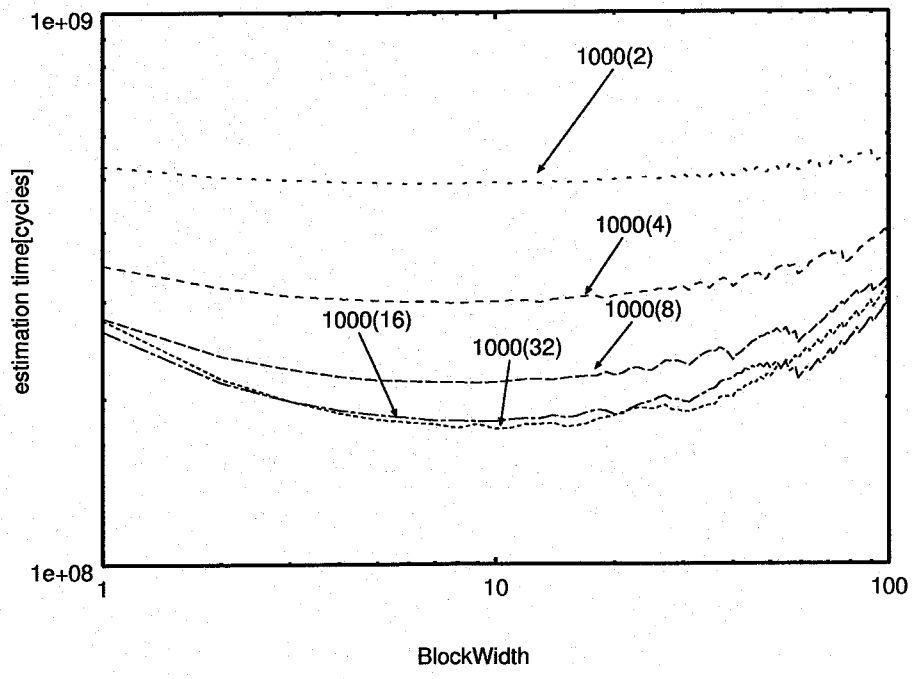


図 3.12: 別のパラメータを用いたスカラプロセッサでの評価結果, 行列サイズ:$N = 1000$ に対して使用するプロセッサ台数を $P = 1, 2, 4, 8, 16, 32$ と仮定した場合. 図中には'N(P)' の形で示している.

VPP300 での実験結果

次にベクトル並列計算機である富士通 VPP300 での実験結果を示す. VPP300 はベクトルプロセッサ単体でのピーク性能が 2.2 GFLOPS で転送速度 570MB/sec のクロスバーネットワークにより接続されたベクトル並列計算機である. SR2201 の場合と同様に ADETRAN4 で記述したプログラムを前章で示した処理系を利用して VPP-FORTRAN に変換して実行した. 変換後の VPP-FORTRAN のソースは約 240 行である.

$N = 100, 500, 1000$ の場合でブロック幅とプロセッサ数を変化させてプロットしたものが図 3.13である. また, $N = 10000$ で通常のものと多段同時消去を行ったものを同時にプロットしたグラフが図 3.14である. まず特徴的なことはプロセッサ数が少ない場合でもブロック化による効果が大きく現われていることである.

全体的に SR2201 のときは横ばいであった曲線が必ず右下がりになっている. また問題サイズが小さい場合でもわずかだが, 並列化の効果が見られる. これは問題サイズが小さくなるとベクトルパイプラインの立ち上がり時間によるペナルティのため遅くなり, 演算部分に対して通信時間が無視できるためである.

N が 1000 程度までは最適なブロック幅は 10 から 50 程度に設定でき, N が大きくなるとブロック幅が小さくなるのは SR2201 と同様である. 次に 4 段の多段消去を行うと約 2 倍近い性能向上が見られる. 特に, $N = 10000$ において $P = 4, m = 40$ の実行結果では 7.1GFLOPS とピーク性能 8.8GFLOPS に対して 80% を出すという好結果が得られた. 最適なブロック幅はスカラプロセッサに比べて大きくとるとよく 10 から 100 の間に設定すればよいことが実験結果から読みとれる. 図 3.13, 図 3.14から見る限りでは $m = 20$ がよい.

SR2201 の場合と同様に対応する範囲をプロットし直したものが 図 3.15と図 3.16である. 図 3.15はパイプライン最適化なし, 図 3.16は最適化ありに対応する. これらから, 図 3.13 は図 3.15 よりむしろ図 3.16に酷似しており, VPP300 のベクトル化コンパイラは多重ループのパイプライン最適化を行なっていることが分かる.

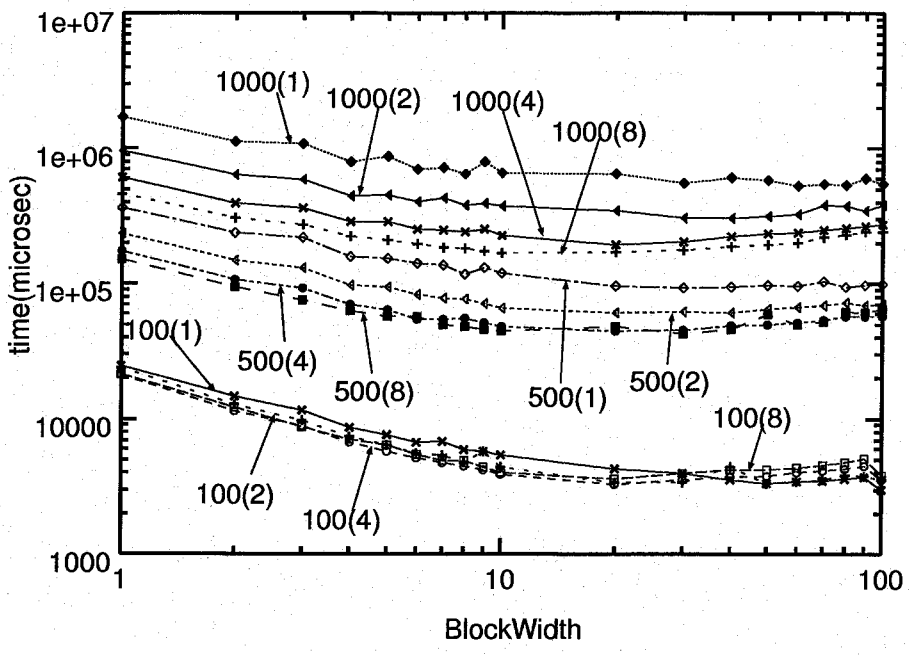


図 3.13: VPP300 での実験結果, 行列サイズ: $N = 10, 100, 500$ に対してプロセッサ台数:$P = 1, 2, 4, 8$ で実行した場合. 図中には'N(P)' の形で示している.

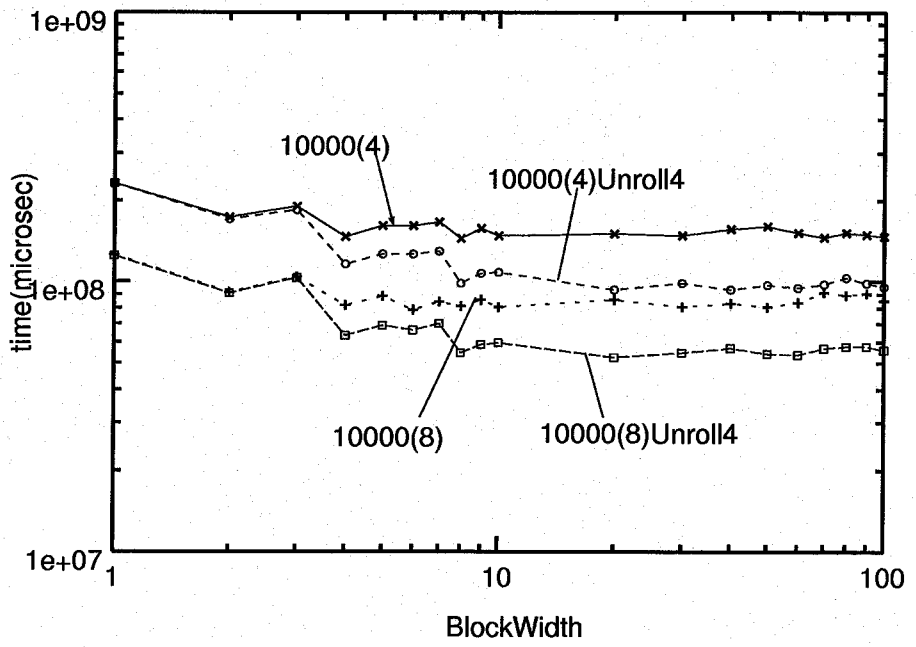


図 3.14: VPP300 での実験結果 (通常版と 4 段アンロール版), 行列サイズ:$N = 10000$ に対してプロセッサ台数:$P = 4, 8$ で実行した場合. 図中には'N(P)' の形で示している.

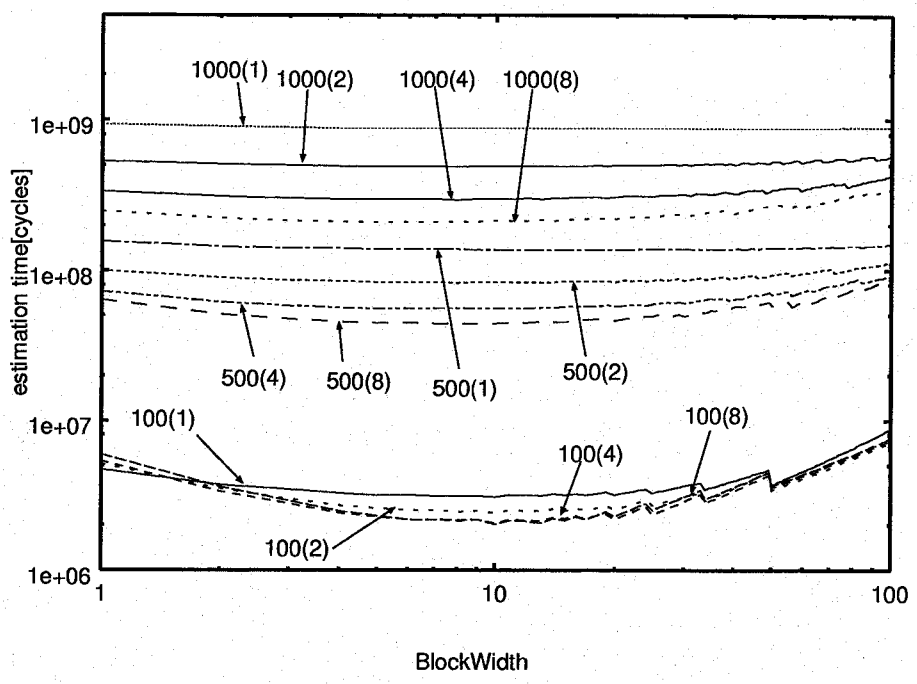


図 3.15: ベクトルプロセッサでの評価結果, 行列サイズ:$N = 100, 500, 1000$ に対して使用するプロセッサ台数を $P = 1, 2, 4, 8$ と仮定した場合. 図中には'N(P)' の形で示している.

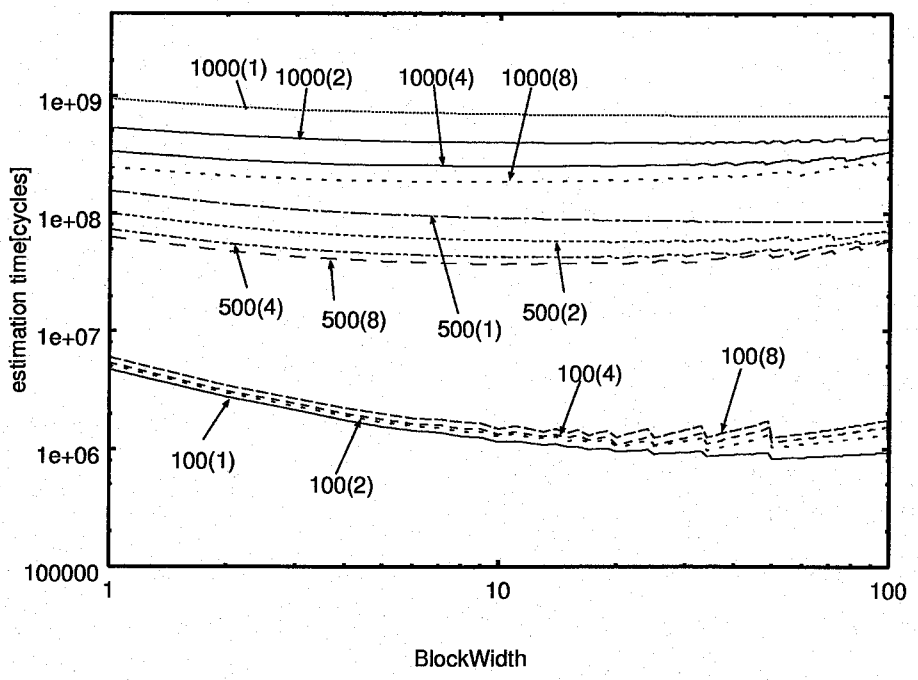


図 3.16: ベクトルプロセッサでパイプライン最適化を想定したときの評価結果, 行列サイズ:$N = 100, 500, 1000$ に対して使用するプロセッサ台数を $P = 1, 2, 4, 8$ と仮定した場合. 図中には'N(P)' の形で示している.

3.2 ブロックハウスホルダー3重対角化法

実対称行列の固有値を計算する際の一手法として, 三重対角行列に相似変換してから固有値ならびに固有ベクトルを計算する方法がとられることが多い. この三重対角化の方法にはギブンスの方法とハウスホルダーの方法の2通りがよく知られている [39, 52, 57, 62]. ギブンスの方法は三重対角化を行う際に, 2×2 の回転行列による相似変換を行い対角および副対角以外の成分を1つづつ0にしていく逐次的な性質を持つ. 一方ハウスホルダーの方法は, 鏡像変換によって1回の変換で1列分 (または1行分) の対角および副対角以外の成分を0化する方法である. とくに, ハウスホルダーの方法はアルゴリズムが, 行列とベクトルの積, 行列の2ランク更新で構成されるためアルゴリズム内の並列性が非常に高いことが知られており, 並列処理を行う場合はハウスホルダーの方法を用いることが一般的になっている.

ハウスホルダーの3重対角化のアルゴリズムを図 3.17に示す.

1: for $i = 1$ to $N-2$
2: determine (β, u) from $A_{(*,i)}$ s.t. $(I - \beta u u^T) A_{(*,i)} = (\underbrace{*, ..., *}_{i+1}, 0, ...)$
3: $w \leftarrow Au$
4: $v \leftarrow w - \beta(w^T u)u$
5: $A \leftarrow A - (uv^T + vu^T)$
6: end

図 3.17: ハウスホルダーの3重対角化

本節では, 上に示したハウスホルダー相似変換の並列化とともにその効率的な実装方法の提案とモデルによる並列化効率の実証を行っていく.

3.2.1 並列ハウスホルダー3重対角化

行列が対称であるときデータを full に保持することは冗長であり下三角または上三角のみに減少させてもよい. その際, 図 3.17の5行目にある rank update 部分を下三角 (または上三角) 部分のみの更新に限定することで計算量を 1/2 にすることができる. 以降, 行列の保持を下三角半分にしたと仮定して議論する.

図 3.17の5行目は各行毎に, プロセッサに割り当てて並列計算可能である. 5行目は行列全体で行う場合には $(i-1)^2$ のコストを各プロセッサで均等に割ることができる. 行列が下半分に保持された場合には, サイクリック分割を行えば全体を保持したものの 1/2 のコストで計算できる.

一方, 3行目の Au の計算部分には変更が加わり計算ステップが変動する. 対称性を使わずに full 行列で計算する場合 Au を axpy で構成する外積型, dot で構成する内積型の2通りで計算する方法が考えられる. 下もしくは上三角のみの要素で行列・ベクトル積を行う例として, LINPACK や BLAS などで用いられている行列の成分を折り返してアクセスする手法が存在する. こ

の手法を axpy 型を元にして構成すると, 次のようなプログラムで記述される.

```
do j=1,n
   do k=1,j-1
     v(k)=v(k)+a(k,j)*u(j)
   enddo
   v(j)=v(j)+a(j,j)*u(j)
   do k=j+1,n
     v(k)=v(k)+a(j,k)*u(j)
   enddo
enddo
```

このプログラムは行列 A の或行成分をアクセスする過程で, 対角部分に到達したら折り返して列成分のアクセスに変更するものである. このプログラムのループは分解, 融合 ならびに 式の順序変更を行うことができ, 次の様なプログラムに書き直すことができる. このプログラムは, 外積形式と内積形式の 2 種類のアクセスを含むことから中間積形式のアクセスと名付けることとする.

```
do j=1,n
   do k=1,j-1
     v(k)=v(k)+a(k,j)*u(j)      ! 1) axpy 型
     v(j)=v(j)+a(k,j)*u(k)      ! 2) dot 型
   enddo
   v(j)=v(j)+a(j,j)*u(j)
enddo
```

このプログラムのループは, 1) v(k)=..., 2) v(j)=... の 2 文に対して, ループインデックス j に関する依存関係 $((j=1) \rightarrow (j=2) \rightarrow \cdots)$ が存在する. 仮にあるプロセッサが担当する 'j' のインデックスが $\{j_1, j_2, ...\}$ であった場合, j_k が昇順 (つまり $j_1 \rightarrow j_2 \rightarrow \cdots, j_1 < j_2 < \cdots,$) に実行されるならば, プロセッサが担当するインデックス部分の割り当て方に関係なく依存関係は満足される. したがって, 上述プログラムのアルゴリズムは最外側ループに関する添字 'j' のインスタンスに関する一次元配列内処理を処理単位とした並列処理で取り扱うことができる.

さらに, 上のプログラムでは変数v にa(k,j)*u() の値を足し併せる代入演算を行っているため, 各プロセッサの最終結果そのものの総和をとらなくてはいけない. 最終的に結果 v(1:n) を全てのプロセッサがその後利用することを想定し, グローバル変数としてのセグメントベクトルで保持することとすると, ADETRAN4 におけるセグメントベクトルの moly 性と reduction 演算を併用して次のように並列化できる.

```
pdo j=1,n
   do k=1,j-1
     v(k)=v(k)+a(k,/j/)*u(j)
     v(j)=v(j)+a(k,/j/)*u(k)
```

```
    enddo
    v(j)=v(j)+a(j,/j/)*u(j)
  pend
  preduce(sum, v(1:n))
```

ここで, 並列化した場合のアルゴリズム全体の計算量は最後の reduction 以外に変化はないことを注意しておく.

次に, 中間積形式とそれ以外の外積, 内積手法とを用いた場合のアルゴリズム全体におけるそれぞれの計算量を比較する. 外積, 内積形式ともに最終的に全てのプロセッサ上に同一の値を保持するために, リダクションまたはギャザーが必要となる.

$$T_{\mathrm{axpy}} = \sum_{n=1}^{N}\Big\{\underbrace{\tau n\left\lceil\frac{n}{P}\right\rceil}_{\text{外積}} + \underbrace{\log P(\xi+\omega n)}_{\text{uのブロードキャスト}} + \underbrace{\log P(\xi+(\omega+\tau)n)}_{\text{wのリダクション}}\Big\} \tag{3.2.1}$$

$$T_{\mathrm{dot}} = \sum_{n=1}^{N}\Big\{\underbrace{\tau n\left\lceil\frac{n}{P}\right\rceil}_{\text{内積}} + \underbrace{\log P(\xi+\omega n)}_{\text{uのブロードキャスト}} + \underbrace{(P-1)(\xi+\omega\lceil n/P\rceil)}_{\text{wのGatherAll転送}}\Big\} \tag{3.2.2}$$

$$T_{\mathrm{mid}} = \sum_{n=1}^{N}\Big\{\underbrace{\sum_{j=1}^{\lceil\frac{n}{P}\rceil}2\tau(n-(j-1)P)}_{\text{中間積(内積+外積)}} + \underbrace{\log P(\xi+\omega n)}_{\text{uのブロードキャスト}} + \underbrace{\log P(\xi+(\omega+\tau)n)}_{\text{wのリダクション}}\Big\} \tag{3.2.3}$$

まず 配列全体のデータを持つ場合の計算手法, T_{axpy} と T_{dot} の比較を行う. 両者の違いは, 上式から判るように w のグローバル変数化を行う部分であり, n に関する 1 次式の係数が異なる. n が小さい部分では $T_{\mathrm{axpy}} < T_{\mathrm{dot}}$ である. 殆んどの範囲で $T_{\mathrm{dot}} < T_{\mathrm{axpy}}$ が成立する. 標準的な並列計算機の例として $\tau = 1, \xi = 100, \omega = 4$ を用いて, $(T_{\mathrm{axpy}} - T_{\mathrm{dot}})/N^2$ についてプロットしたコンター図が図 3.18である. この図からも, dot 方式は実用上の殆んどの領域で優れていることが示される.

次に full 領域のアルゴリズムに dot を使用するものを採用するとして, 下三角領域の更新アルゴリズムとでの計算ステップの差を評価する. A の更新部分と (図 3.17 5行目), Au の計算部分 (図 3.17 3行目) が両者の違いである. A の更新部分の差は

$$\Delta_{\mathrm{update}} = \sum_{n=1}^{N} 2\tau n\left\lceil\frac{n}{P}\right\rceil - \sum_{n=1}^{N}\sum_{j=1}^{\lceil\frac{n}{P}\rceil}2\tau(n-(j-1)P) \sim \frac{\tau}{3P}N^3 + \frac{\tau}{2P}N^2 \tag{3.2.4}$$

である. したがって, Au の計算部分の差 $(T_{\mathrm{dot}} - T_{\mathrm{mid}})$ を加えた値は

$$\begin{aligned}\Delta &\sim \frac{\tau}{3P}N^3 + \frac{\tau}{2P}N^2 + \frac{P-1}{2P}\omega N^2 - \frac{1}{2}\log P(\omega+\tau)N^2 + \xi(P-1)N - \xi\log PN \\ &= \frac{\tau}{3P}N^3 + \left\{\frac{\tau}{2P} + \frac{P-1}{2P}\omega - \frac{1}{2}\log P(\omega+\tau)\right\}N^2 + \xi(P-1+\log P)N \end{aligned} \tag{3.2.5}$$

となる. 上式 Δ が十分大きな正の値であれば, 対称性を利用した中間積アルゴリズムが有効であることの指標となる.

この指標を用いて, 標準的な並列計算機の例として$\tau = 1, \xi = 100, \omega = 4$のもとで$\Delta$をコンタープロットしたものを図3.19に示す. 問題サイズが小さい場合に, 多くのプロセッサを利用すると中間積が劣ることが読み取れる. しかし, 図にあるような$N = 2000$程度の問題でプロセッサを100以上利用することは考えにくい.

また, 実並列計算機上で内積アルゴリズムと中間積アルゴリズムを用いてハウスホルダー変換を行った結果の両者での計算時間の差を図3.20と図3.21に示す. 両グラフには (内積アルゴリズム − 中間積アルゴリズム) をプロットしている. 計算に使用するプロセッサ数を増加させると, 全体の計算時間が減少するためグラフの傾きが緩やかになるがその傾向はΔをコンタープロットした図3.19と同様である. また, 使用可能なプロセッサ台数が多くないため事実を観測できないが, 更にプロセッサ台数を増やした場合には通信の効果によってグラフ全体が下に落ち込むことになるため, 問題サイズが小さい領域では内積アルゴリズムが高速になるという結果も容易に想像できる. これは, 図3.19の左上部分に相当するがこの領域で内積と中間積との優位な性能差があるとは言いがたい範囲にある.

以上, 論理的推測, 実計算機での実測から, ハウスホルダー三重対角化アルゴリズムにおいて実用上の問題サイズならびにプロセッサ数では対称性を活かした本アルゴリズムは有効であることが示された.

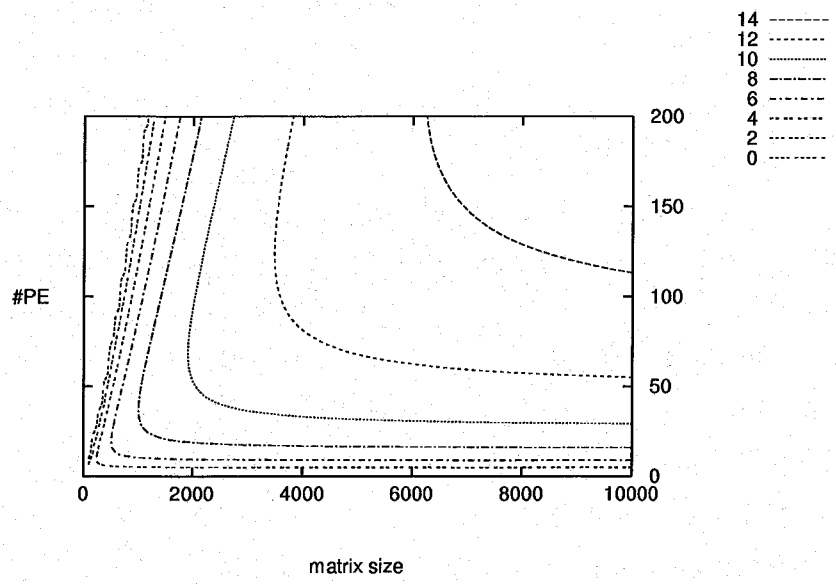


図 3.18: T_{dot}とT_{axpy}の比較: 行列サイズ (横軸) と使用するプロセッサ数 (縦軸) における $(T_{\mathrm{axpy}} - T_{\mathrm{dot}})/N^2$ の値を示している.

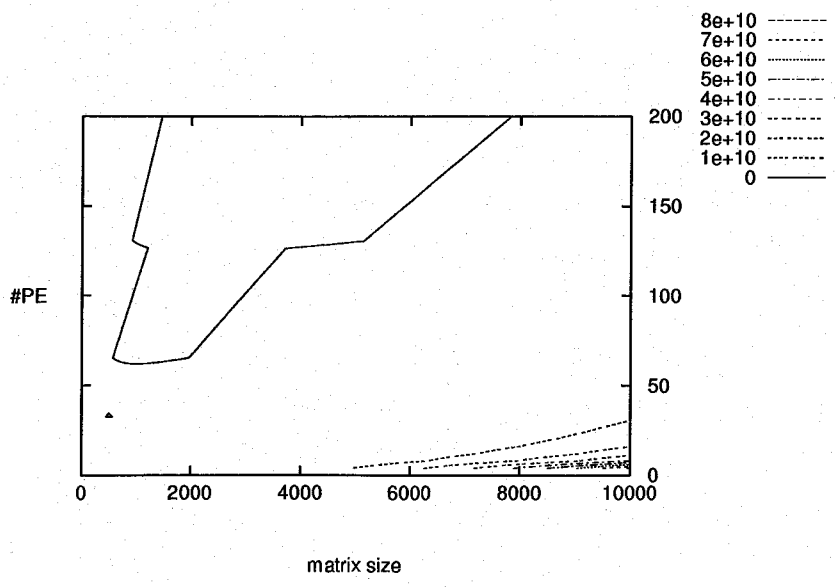


図 3.19: 中間積アルゴリズムと内積アルゴリズムの比較: 行列サイズ (横軸) と使用するプロセッサ数 (縦軸) における Δ(式 3.2.5を参照) の値を示している.

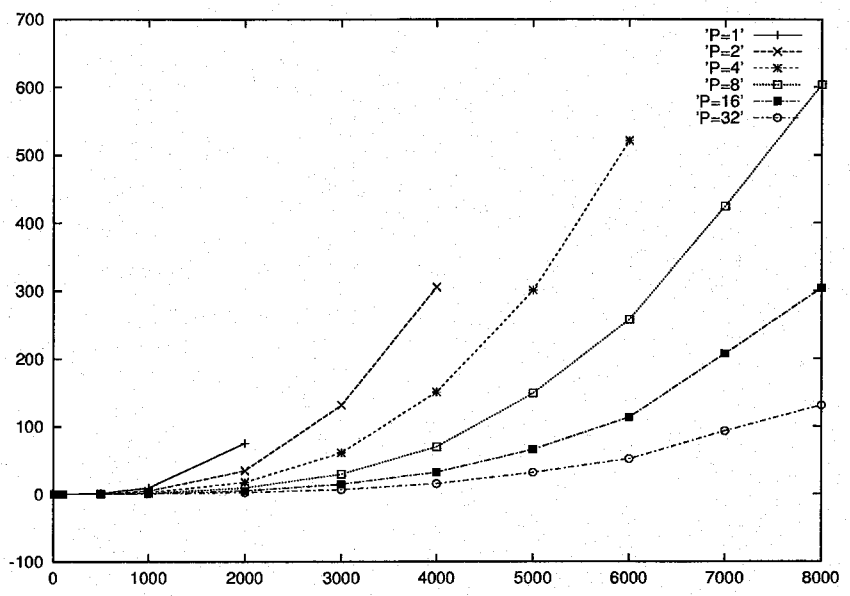


図 3.20: 中間積アルゴリズムと内積アルゴリズムの並列計算機 SR2201 での実計算時間での比較: 行列サイズ (横軸) と, 計算時間の差 '内積アルゴリズム − 中間積アルゴリズム' (縦軸, 単位は秒) をプロットしている.

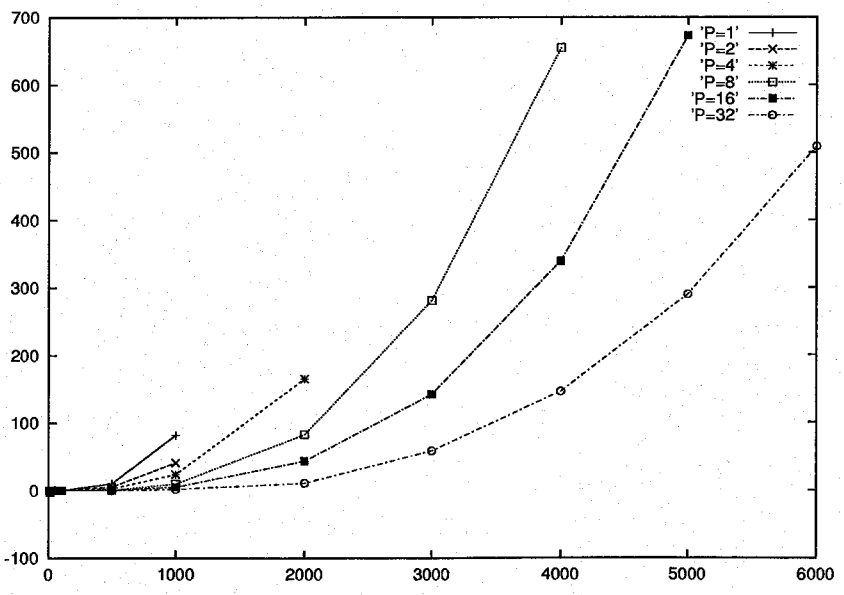


図 3.21: 中間積アルゴリズムと内積アルゴリズムの並列計算機 SP/2 での実計算時間での比較: 行列サイズ (横軸) と, 計算時間の差 '内積アルゴリズム − 中間積アルゴリズム' (縦軸, 単位は秒) をプロットしている.

3.2.2 並列ハウスホルダー3重対角化ブロック版

本節ではDongarraらが示した, ブロックアルゴリズムに関してモデル化を行い計算ステップの評価によりその有効性を議論する.

Dongarraら [15] の示したブロックアルゴリズムは以下のようになる. ただし彼らの手法はパネル分割に密接していたが, 以下のものは中間変数として W を利用しているためデータ分割にアルゴリズムが直接影響を受けない点が異なる. なお, 本図で示したアルゴリズムをADETRAN4で記述したプログラムを付録Bに示している. 同プログラムは本節の後半部分で示す数値実験でも利用されているものである.

1: for $i = 0$ to $N - m$ step by m
2: $W_{(*,1:m)} \leftarrow A_{(*,i+1:i+m)}, U \leftarrow \emptyset, V \leftarrow \emptyset$
3: for $j = 1$ to m
4: determine (β, u) from $W_{(*,j)}$ s.t. $(I - \beta u u^T) W_{(*,j)} = (\underbrace{*, ..., *}_{i+j+1}, 0, ...)$
5: $w \leftarrow (UV^T - VU^T)u$
6: $w \leftarrow Au - w$
7: $v \leftarrow w - \beta(w^T u)u$
8: $U \leftarrow [U\ u], V \leftarrow [V\ v]$
9: $W_{(*,j+1:m)} \leftarrow W_{(*,j+1:m)} - (uv^T + vu^T)_{(*,j+1:m)}$
10: end
11: $A_{(*,i+1:i+m)} \leftarrow W_{(*,1:m)}$
12: $A_{(*,i+m+1:n)} \leftarrow A_{(*,i+m+1:n)} - (UV^T + VU^T)$
13: end

図 3.22: ブロックハウスホルダー3重対角化アルゴリズム

5行目が逐次処理として, また9行目が従来の並列処理から孤立したものとして, 非ブロック化アルゴリズムと比較した際に追加される処理である. 非ブロック化部分を先節で示した中間積アルゴリズムを用いるとして, 追加された部分の計算量はそれぞれ以下の様に求められる.

$$T_1 = \sum_{i=0}^{\lceil \frac{N-m}{m} \rceil} \sum_{j=1}^{m} \sum_{k=1}^{j-1} (2\delta + 7\tau(N - im)) \tag{3.2.6}$$

$$\sim \frac{7}{4}\tau(m-1)N^2 + \left(\delta - \frac{7}{4}\tau m\right)(m-1)N \tag{3.2.7}$$

$$T_2 = \sum_{i=0}^{\lceil \frac{N-m}{m} \rceil} \sum_{j=1}^{m} \sum_{k=j+1}^{m} (\delta + 4\tau(N - im))\frac{1}{P} \tag{3.2.8}$$

$$\sim \left(2\tau(m-1)N^2 + (\delta - 2\tau m)(m-1)N\right)\frac{1}{2P} \tag{3.2.9}$$

したがって

$$T_1+T_2 \sim \left(\frac{7}{4}+\frac{1}{P}\right)\tau(m-1)N^2+\left(\left(1+\frac{1}{2P}\right)\delta-\left(\frac{7}{4}+\frac{1}{P}\right)\tau m\right)(m-1)N \tag{3.2.10}$$

の計算ステップ分が増加したことになる。

後半のループの演算量はブロック化のため表記上若干増加していると見ることができるが, 0 成分を考慮すると実質の計算量増分はない. 全体の演算量そのものは T_1+T_2 分増加しており決して減少しているわけではないので, アルゴリズム全体で高速化がなされるのはブロック演算を行うことによる演算の面積体積効果に因るところが大きいと云える. 実際高速化の手法としてキャッシュを意識したタイリングやループアンローリングによってメモリレジスタ間のデータの移動を減少させる手法が存在し, Dongarra らの手法ではそれが高速化の主要因であるとの解釈がなされている.

今, ブロック化された後半ループ部分のステップ数の評価を行う. 後半のループの演算ステップは

$$T_3=\sum_{i=0}^{\lceil\frac{N-m}{m}\rceil}\sum_{j=1}^{m}\left\lceil\frac{N-im}{P}\right\rceil(\delta+4\tau(N-im)) \tag{3.2.11}$$

である. ブロックの回転に対する (上の式では 'j') l 段のループアンローリングを行った場合の, 演算ステップを求めると次の様になる.

$$T_3(l)=\sum_{i=0}^{\lceil\frac{N-m}{m}\rceil}\left\lceil\frac{N-im}{P}\right\rceil(\delta+4\tau l(N-im)\alpha(l))\left\lceil\frac{m}{l}\right\rceil \tag{3.2.12}$$

ここで l : ループアンローリング段数 $(1\le l\le m)$, δ : ループオーバヘッド, α : アンローリング加速率 (≤ 1) である.

最後の更新式 (3.2.12) に対して $\lceil\cdot\rceil$ などを開き, 簡略化した式に書き直す. ここで, 4τ は計算コストがメモリ – レジスタ間のデータ転送に支配されることから算出されるものである. 本ループの場合, 定数要素のロード除きロード 3, ストア 1 で 1 行列要素の更新ができることから 4τ と決まる. さらに, このループでは l 段アンローリングを行うことにより, 全体としてロード $(2l+1)/l$, ストア $1/l$ で 1 行列要素の更新が可能となり, 結果 $\alpha(l)=(l+1)/2l$ と求められる $(\lim_{l\to\infty}\alpha(l)=1/2)$.

$$T_3(l)=\frac{1}{2}\sum_{i=0}^{\lceil\frac{N-m}{m}\rceil}\frac{N-im}{P}\left(\delta+4\tau l(N-im)\frac{l+1}{2l}\right)\frac{m}{l} \tag{3.2.13}$$

$$\sim\frac{(l+1)\tau}{3Pl}N^3+\left(\frac{\delta}{4Pl}+\frac{(l+1)m\tau}{2Pl}\right)N^2 \tag{3.2.14}$$

ここで, ブロックアルゴリズムが非ブロックアルゴリズムに対して高速であるための必要条件が T_1,T_2,T_3 の関係式より導かれる. つまり, 次に示す D について $D\gg 0$ であることが必要である.

$$D=\underbrace{\left(\,T_3(1)|_{m=1}-T_3(l)\,\right)}_{\text{アンローリングによる加速分}}-\underbrace{\left(\,T_1+T_2\,\right)}_{\text{ブロック化による増分}} \tag{3.2.15}$$

一般的にループアンローリングは4段程度行うことが多い．$l=4$ として $\delta=100, \tau=1$ のとき D は次の様になる．

$$D \sim \frac{2}{3P}N^3 + \frac{26}{P}N^2 - \frac{5}{12P}N^3 - \frac{50+5m}{8P}N^2 - \left(\frac{7}{4}+\frac{1}{P}\right)(m-1)N^2 \quad (3.2.16)$$

$$= (2N + 166 - 3m - 14(m-1)P)\frac{N^2}{8P} \quad (3.2.17)$$

上式 (3.2.17) よりブロック幅 m やプロセッサ数 P が過剰に大きい場合には，$D >> 0$ でなくなるためブロック化による逆効果が生じる．また，本ブロック化アルゴリズムにおいて最も効率的なブロック幅は，アンローリングの段数に一致するもしくはその数倍程度と予測できる．

実際，実機 SR2201 上でブロック化部分に対する4段のループアンローリングを行い，その測定結果を図 3.23, 3.24, 3.25, 3.26に示す．この結果からも分かるように，ブロック化による効果が優位に現れる領域はループアンローリング段数の数倍程度までである．また，問題サイズが比較的小さい N=1000 程度では，ブロック幅が極端に大きくなったときに負荷均衡がとれなくなり，かえってブロック化を行わないものよりも遅くなるという結果が得られている．これは，式 (3.2.17) での第一因子内の N の効果を示しており，N が 1000, 2000, 4000, 8000 と増大するごとに右上がりのカーブが緩やかになる事実ならびに，ブロック化による高速化率が $P=8$ を基準とした場合に $N=1000$ で約 20%, $N=2000$ で約 25%, $N=4000$ で約 30% と伸びていく事実の両者を的確に表現する項である．

これら観測結果とそれらに対する考察からも，本節で議論したブロック化の有効性とその範囲評価の正当性が示される．

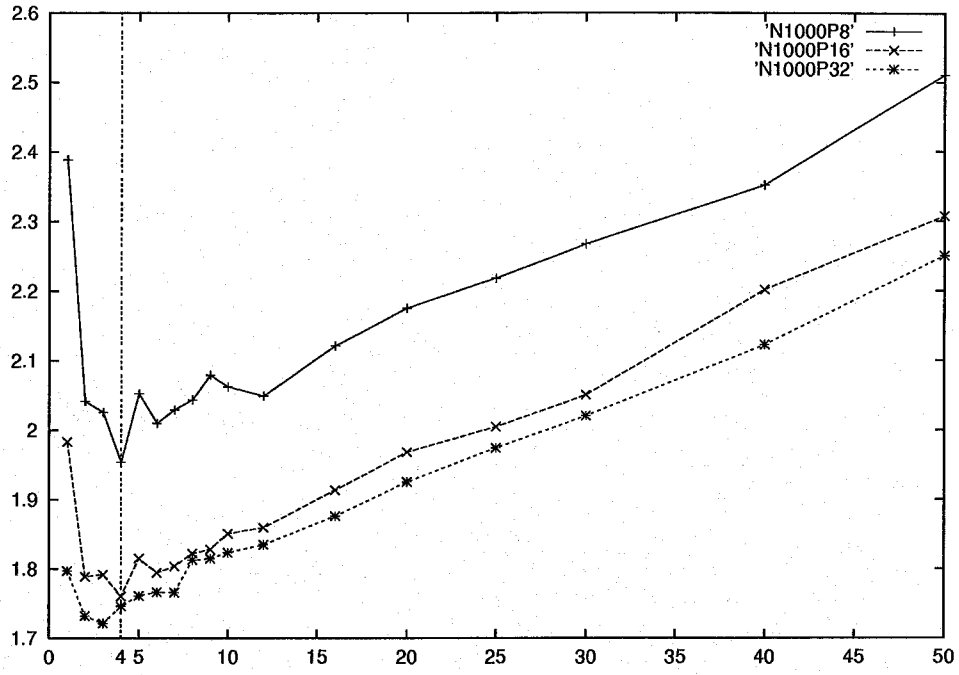


図 3.23: 並列計算機 SR2201 でのブロック化の効果測定時間 (行列サイズ 1000×1000): ブロック幅 (横軸) と, 計算時間 (縦軸, 単位は秒) をプロットしている (実線は PE 数 8, 破線は PE 数 16, 点線は PE 数 32).

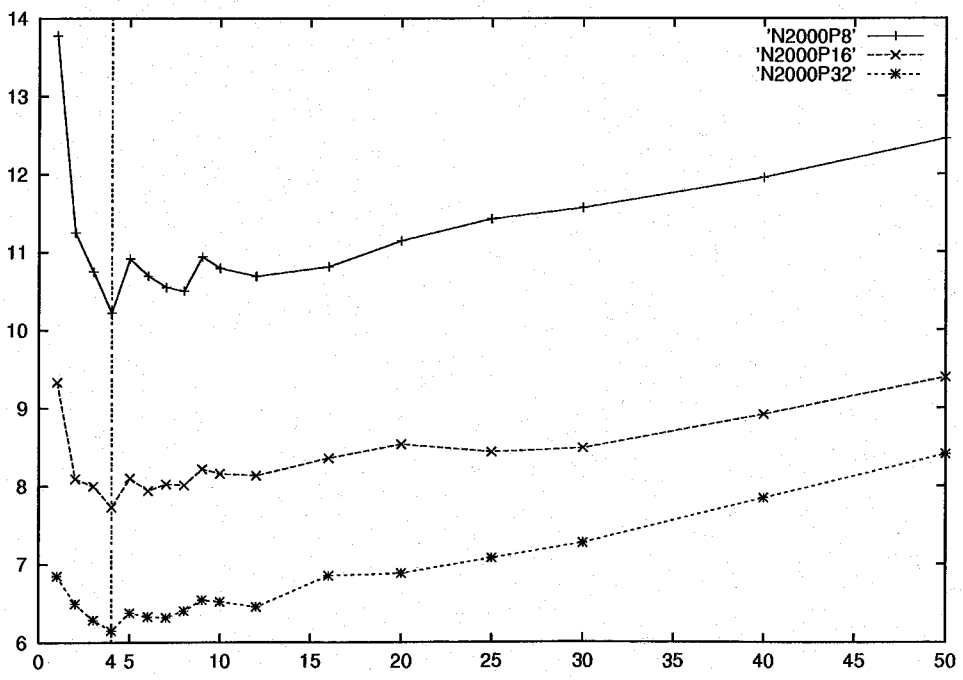


図 3.24: 並列計算機 SR2201 でのブロック化の効果測定時間 (行列サイズ 2000×2000): ブロック幅 (横軸) と, 計算時間 (縦軸, 単位は秒) をプロットしている.(実線は PE 数 8, 破線は PE 数 16, 点線は PE 数 32).

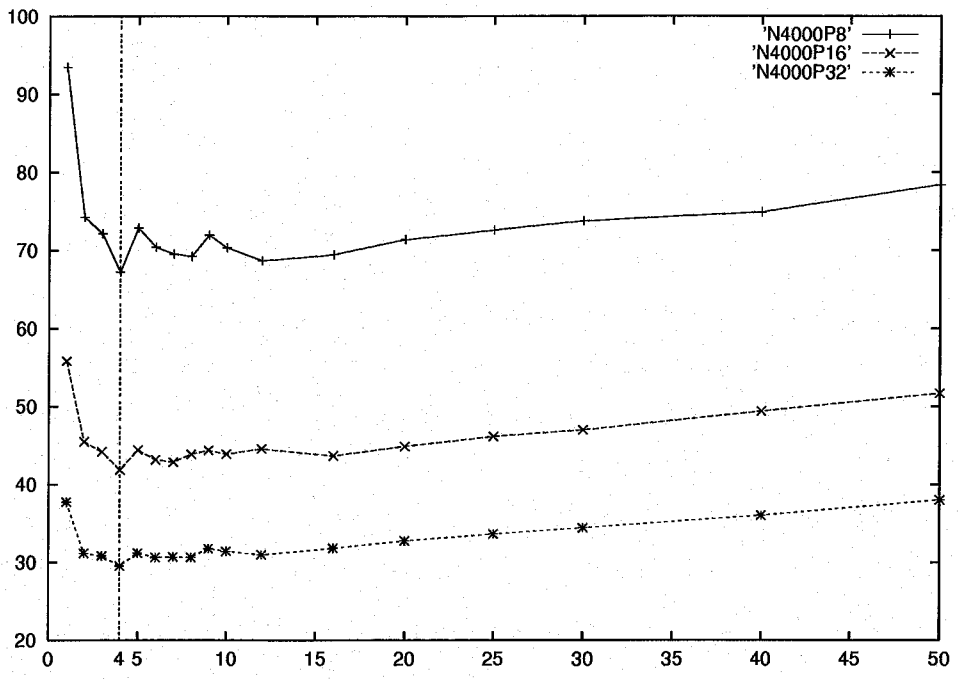


図 3.25: 並列計算機 SR2201 でのブロック化の効果測定時間 (行列サイズ 4000×4000): ブロック幅 (横軸) と, 計算時間 (縦軸, 単位は秒) をプロットしている (実線は PE 数 8, 破線は PE 数 16, 点線は PE 数 32).

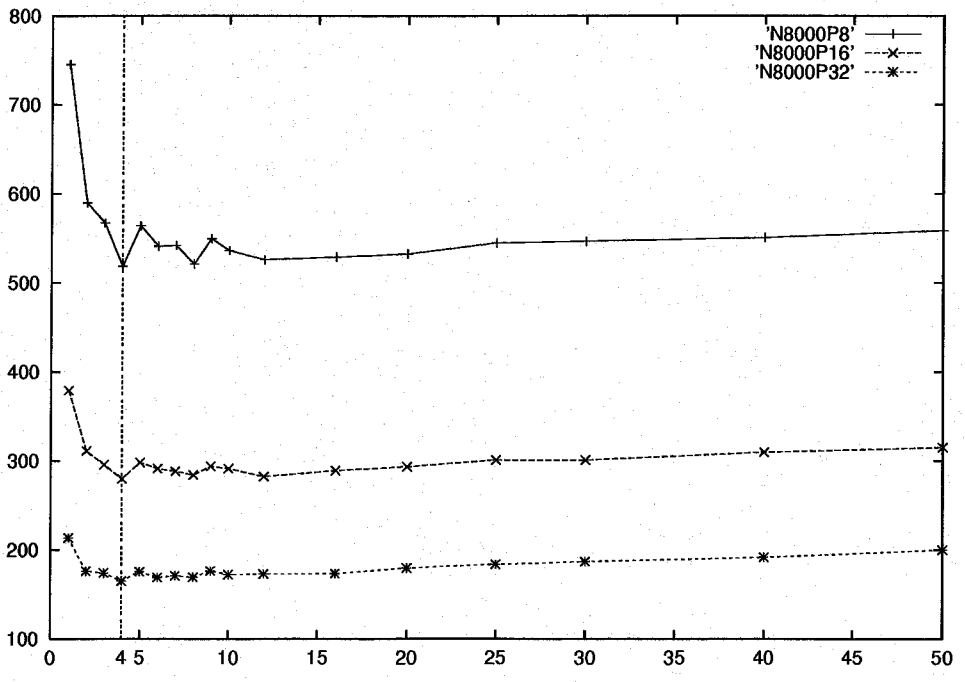


図 3.26: 並列計算機 SR2201 でのブロック化の効果測定時間 (行列サイズ 8000×8000): ブロック幅 (横軸) と, 計算時間 (縦軸, 単位は秒) をプロットしている.(実線は PE 数 8, 破線は PE 数 16, 点線は PE 数 32).

3.3 Divide and conquer 法

3.3.1 Divide and conquer 法の概要

実対称 3 重対角行列 T について,

$$T = \begin{pmatrix} \alpha_1 & \beta_1 & & \\ \beta_1 & \alpha_2 & \beta_2 & \\ & \ddots & \ddots & \ddots \\ & & \beta_{n-1} & \alpha_n \end{pmatrix} \tag{3.3.1}$$

を次の様に分解する.

$$T = \left(\begin{array}{c|c} T_1 & \\ \hline & T_2 \end{array}\right) + \rho w w^T = \tilde{T} + \rho w w^T \tag{3.3.2}$$

$$w = (\underbrace{0 \ldots 1}_{k} | \underbrace{1 \ldots 0}_{n-k})^T, \quad \rho = \beta_k \tag{3.3.3}$$

$$T_1 = \begin{pmatrix} \alpha_1 & \beta_1 & & \\ \ddots & \ddots & \ddots & \\ \beta_{k-2} & \alpha_{k-1} & \beta_{k-1} & \\ & \beta_{k-1} & \alpha_k - \beta_k & \end{pmatrix} \tag{3.3.4}$$

$$T_2 = \begin{pmatrix} \alpha_{k+1} - \beta_k & \beta_{k+1} & & \\ \beta_{k+1} & \alpha_{k+2} & \beta_{k+2} & \\ & \ddots & \ddots & \ddots \\ & & \beta_{n-1} & \alpha_n \end{pmatrix} \tag{3.3.5}$$

ここで, T_1, T_2 の行列に対して固有値・固有ベクトルが既値として与えられていると仮定する. そして T_1 の固有値を昇順に並べて $d_{11}, d_{12}, \ldots$ と表し, 対応する固有ベクトルを $q_{11}, q_{12}, \ldots$ とする. また T_2 の固有値を昇順に並べて $d_{21}, d_{22}, \ldots$ と表し, 対応する固有ベクトルを $q_{21}, q_{22}, \ldots$ とする. さらに $d_{ij}(i = 1, 2)$ を併せて, 昇順に並べ替えたものを改めて $d_1, d_2 \ldots$ とし,

$$D = \mathrm{diag}(d_1, d_2, \ldots) \tag{3.3.6}$$

$$Q = (q_1, q_2, \ldots) \tag{3.3.7}$$

$$z = Q^T w \tag{3.3.8}$$

を定義する。このとき, 行列 T の固有値は $Q^T(\tilde{T} + \rho w w^T)Q = D + \rho z z^T$ の固有値に等しいことを注意しておく. また T の固有値を昇順に並べ替えたものを $\lambda_1, \lambda_2, \ldots$ とし, 対応する固有ベクトルを $x_1, x_2, \ldots$ と表すこととする.

次に, T の固有値 λ に対応する固有ベクトルを x とすると式 (3.3.2) から

$$Tx = \lambda x = \tilde{T}x + \rho(w^T x)w \tag{3.3.9}$$

が成立する．上式に Q^T を左から作用させると，

$$\lambda\tilde{x} = D\tilde{x} + \rho(z^T\tilde{x})z \tag{3.3.10}$$

が成立する．ただし，$\tilde{x} = Q^T x$ である．上式を整理すると，

$$\tilde{x} = \rho(z^T\tilde{x})(\lambda I - D)^{-1}z \tag{3.3.11}$$

となる．さらに z^T を左から乗じて，$z^T\tilde{x} \neq 0$ を用いれば

$$1 - \rho z^T(\lambda - D)^{-1}z = 0 \tag{3.3.12}$$

なる有理方程式が得られる．式 (3.3.12) の根は，行列 T の固有値に一致する．

更に，式 (3.3.11) から $\tilde{x}$ は $(\lambda I - D)^{-1}z$ のスカラ倍であることから，有理方程式の根 λ を代入することで $\tilde{x}$ の主要成分比 $\tilde{x}'$ が求められる．続けて正規化と Q を作用することで T の固有ベクトル x が得られる．

$$\tilde{x}' = (\lambda I - D)^{-1}z \tag{3.3.13}$$

$$= \left(\frac{z_1}{\lambda_1 - d_1}, \frac{z_2}{\lambda_2 - d_2}, \ldots, \frac{z_n}{\lambda_n - d_n}\right)^T \tag{3.3.14}$$

$$x = \frac{1}{\|\tilde{x}'\|}Q\tilde{x}' \tag{3.3.15}$$

この一連の操作により行列 T の固有値，固有ベクトルが計算される．つまり，行列 T の固有値と固有ベクトルは，T_1, T_2 の固有値と固有ベクトルから得られる有理方程式 (3.3.12) と式 (3.3.14), (3.3.15) を解くことで得ることができる．小問題 T_1, T_2 はそれぞれ独立しているため，それらに対してさらに同様の手法を適用することにより再帰的に小問題に分けていくことができる．そして小問題の結果が上位の問題に戻されていく．そういった意味で，この固有値求解手法は divide and conquer[9] (以下 DC 法または考案者の名をとり Cuppen's DC と呼ぶ) と呼ばれている．

3.3.2 Secular 方程式の性質について

有理方程式 (3.3.12) を，具体的に書き表すと

$$1 - \rho\sum_i \frac{z_i^2}{\lambda - d_i} = 0 \tag{3.3.16}$$

となる．この方程式を **Secular 方程式**という．式 (3.3.16) の左辺を λ の関数とみなし $f(\lambda)$ とおき，関数 $f(\lambda)$ の性質について本節で述べる．

始めに，仮定として $\rho > 0$、$z_i \neq 0$ かつ d_j が全て異なるとする．このとき，d_i は 1 次の極になり，十分小さな $\varepsilon(>0)$ に対して $f(d_i - \varepsilon) < 0$ かつ $f(d_{i+1} + \varepsilon) > 0$ が成立する．また，

$$f'(\lambda) = \rho\sum_{i=1}^{n} \frac{z_i^2}{(\lambda - d_i)^2} > 0 \quad \text{for all } x\ (\neq d_j, j = 1, \ldots, n) \tag{3.3.17}$$

より, 区間 (d_i, d_{i+1}) について考えると, f が単調増加であるから区間内に 1 つ $f(\lambda) = 0$ の根が存在する. これを $i = 1, 2, \ldots, n$ に適用すると $f(\lambda) = 0$ の根の存在範囲は

$$d_1 < \lambda_1 < d_2 < \ldots < \lambda_{n-1} < d_n < \lambda_n \tag{3.3.18}$$

となる. さらに式 (3.3.2) の トレースをとると

$$\sum_{i=1}^{n} \lambda_i = \sum_{i=1}^{n} d_i + 2\rho \tag{3.3.19}$$

なる関係を得る. 式 (3.3.18) から $\lambda_i - d_i > 0$ であるので,

$$\sum_{i=1}^{n-1} (\lambda_i - d_i) = d_i + 2\rho - \lambda_i > 0 \tag{3.3.20}$$

となり, 次式

$$\lambda_n < d_n + 2\rho \tag{3.3.21}$$

が成立する. したがって, 全ての根 (つまり固有値) の存在範囲が次の様に特定される.

$$d_1 < \lambda_1 < d_2 < \ldots < \lambda_{i-1} < d_n < \lambda_n < d_n + 2\rho \tag{3.3.22}$$

secular 方程式の全ての根の存在範囲が特定されるので, 2 分法や Newton 法などによって求めることができる.

Deflation step

前段において $z_i \neq 0$ と d_i は重複しないことの 2 点を仮定したが, 仮定を満足しない場合に自明な解が存在し secular 方程式から取り除くことができる deflation 操作について説明する.

今 $z_i = 0$ となる成分 i が存在する場合, 固有値は d_i であり, 固有ベクトルは $(0, \ldots, \overset{i}{\overset{\vee}{1}}, \ldots, 0)$ と直ちに求められる. 実際、式 (3.3.10) にこれらを代入すると

$$\left(\mathrm{diag}(d_1, d_2, \ldots, d_n) + \rho z z^T\right) \begin{pmatrix} \vdots \\ 1 \\ \vdots \end{pmatrix} \leftarrow i = \begin{pmatrix} \vdots \\ d_i \\ \vdots \end{pmatrix} \leftarrow i + \rho z 0 = d_i \begin{pmatrix} \vdots \\ 1 \\ \vdots \end{pmatrix} \leftarrow i \tag{3.3.23}$$

が成立し, 固有値ならびに固有ベクトルになっていることが確認できる. したがって, i 行および i 列成分を落した問題を考察すれば十分ということになる. この様に $z_i = 0$ の部分を走査し問題から切り放す操作を deflation という.

次にもう一つの条件である, $d_i = d_j, (i \neq j)$ となる場合, つまり, T_1 と T_2 の固有値の組の中に重複するものが存在する場合を考える. このとき, i ならびに j 番目の行及び列に関する 2×2 小行列に対する適当な回転操作 $U(\theta)$ を行なうことにより, $\tilde{z}_j = 0$ とすることができる.

$$U(\theta)(D + \rho z z^T)U(-\theta)$$

$$
\begin{aligned}
&= \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \left(\begin{pmatrix} d_i & \\ & d_j \end{pmatrix} + \rho \begin{pmatrix} z_i \\ z_j \end{pmatrix} (z_i \;\; z_j) \right) \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \\
&= \begin{pmatrix} d_i & \\ & d_j \end{pmatrix} + \rho \begin{pmatrix} \tilde{z}_i \\ 0 \end{pmatrix} (\tilde{z}_i \;\; 0)
\end{aligned}
\tag{3.3.24}
$$

ここで, $\tilde{z}_i = (z_i^2 + z_j^2)^{\frac{1}{2}}, \cos\theta = z_i/\tilde{z}_i, \sin\theta = z_j/\tilde{z}_i$. この回転操作を施すことにより, j 番目の要素が 0 となり前述の deflation 操作が適用可能となる. 先の deflation 操作の説明から, d_i が固有値になっていることが分かる.

また, この操作は再帰的に行うことが可能なので, l 個の d_k の値が重複している場合には $l-1$ 回の deflation 操作を行うことができるとともに, d_k が行列 T においても $l-1$ 重に重複している固有値となることを意味する. これは逆も正しく, 行列 T の固有値 λ が $l-1$ 重に重複している場合は, $d_k = \lambda$ が成立し, d_k は l 重に重複する.

これにより, 行列 T が重複する固有値を通常, 逆反復法などで注意を払わなくてはならなかった重複固有値計算の直交化操作などを解消することができる.

3.3.3 並列 DC 法について

以上のように DC 法とは 2 分割した問題から得られる情報を利用して, 通常の計算法よりもより少ない計算量で固有値や固有ベクトルを算出しようとするものである. 当然分割された小問題に対してさらに DC 法を適用することができる. この操作を再帰的に行うと, 元の問題を根とする 2 分木状の問題を構成することになる (図 3.27). このように複数に分けられた小問題の固有値求解同士は独立であり自明な並列性を有している. したがって, DC 法は効率的な並列処理が期待できる手法といえる.

次に DC 法を並列に計算する場合の, プロセッサの割り当てやデータの分散について考察していく. 各小問題が自明な並列性を有している点を利用すると, 個々の小問題にプロセッサを割り当てて計算を行なうのが自然な並列化といえる. しかし, 下位に位置する問題では十分な数の小問題が存在するため多くのプロセッサを用いて並列計算が可能であるが, 上位の問題に行くに従い問題数が少なくなるために少数のプロセッサしか使用できなくなる. また, 上位の問題は上位に行くに従い計算負荷が大きくなるため, 並列計算の効果を十分に得ることができない.

下位の問題に限らず上位の問題に対しても, 計算負荷を均一にし良好な並列化効率を得るプロセッサの割り付け方法を考案する. secular 方程式の根の求解は各段位の問題において, 固有値の個数の合計は一定であるので問題全体では十分な並列性を有している. したがって, 各段位での固有値に着目し, secular 方程式での求解をプロセッサに割り当てることで全ての段位での計算負荷を均一に保つことができる. 図 3.27の 2 分木の sub(sub..)problem に示した様な, 各レベルでのプロセッサマッピングが一例として挙げられる (図は Block 分割に相当するが, 同様に Cyclic 分割も考えることができる). これにより, 上位の問題に対しても, 下位問題に対しても, 複数のプロセッサを割り当てることで負荷分散が実現される.

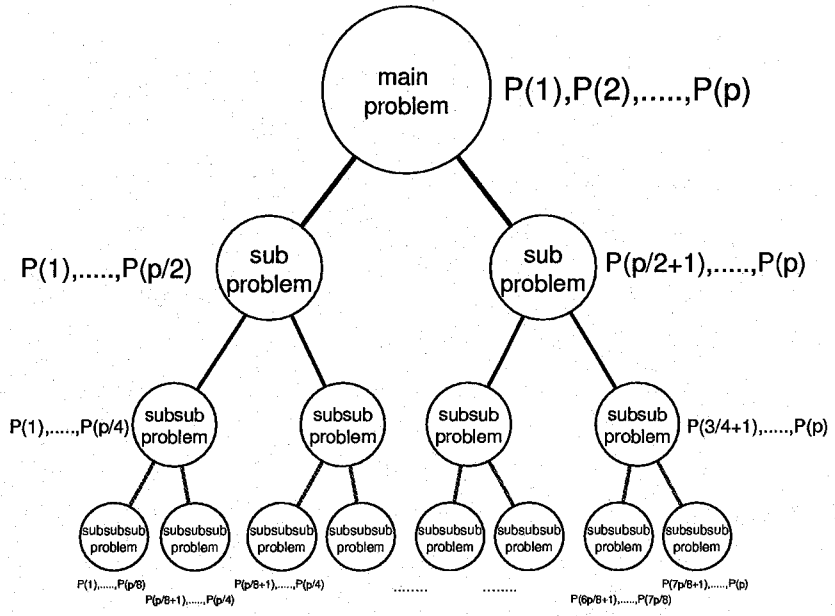


図 3.27: 問題の分割と 2 分木

3.3.4 HDC 法を用いた固有値探索

Cuppen の DC 法は, 行列 T を 2 つにブロック分割する方法である. 小問題での計算をさらに少なくする試みとして, Arbenz[3] の restricted rank modification をもとにした 3 つのブロックに分割する方法を提案した.

3 重対角行列 T を次のように分割する。

$$T = \left(\begin{array}{c|c|c} T_1 & & \\ \hline & T_2 & \\ \hline & & T_3 \end{array}\right) + VV^T \tag{3.3.25}$$

$$V = \left(\begin{array}{c|c|c} 0\cdots\rho_1^{1/2} & \rho_1^{1/2}\cdots 0 & \\ \hline & 0\cdots\rho_2^{1/2} & \rho_2^{1/2}\cdots 0 \end{array}\right)^T \tag{3.3.26}$$

いま T_1, T_2, T_3 の固有値 $d_{i1}, d_{i2}, \ldots$, (i=1,2,3) ならびに固有ベクトル $q_{i1}, q_{i2}, \ldots$, (i=1,2,3) が既値とする. 固有値 d_{ij} を昇順に並び替え, 改めて $d_1, d_2, \ldots, d_n$ と書くこととする. 対応する固有ベクトル $q_1, q_2, \ldots, q_n$ として, 次式の様にその順に並べて作られる行列を Q とおく.

$$Q^T \begin{pmatrix} T_1 & & \\ & T_2 & \\ & & T_3 \end{pmatrix} Q = \mathrm{diag}(d_1, d_2, \ldots, d_n) = D \tag{3.3.27}$$

上式 (3.3.25) の modified Weinstein determinant は, Arbenz らの表現を用いると次式の様になる.

$$f(\lambda) = \det(I_2 - V^T(\lambda - T_1 \oplus T_2 \oplus T_3 \oplus)^{-1} V) \tag{3.3.28}$$

$$= \det(I_2 - U^T(\lambda - D)^{-1}U) \tag{3.3.29}$$

$$= 1 + \sum_{k=1}^{r(=2)} (-1)^k \sum_{\substack{1\le\tau_1<\cdots<\tau_k\le r \\ 1\le j_1<\cdots<j_k\le n}} \frac{(\det U_{j_1\cdots j_k;\tau_1\cdots\tau_k})^2}{(\lambda - d_{j_1})\cdots(\lambda - d_{j_k})} \tag{3.3.30}$$

ここで $U = Q^T V$ であり

$$U_{j_1\cdots j_k;\tau_1\cdots\tau_k} = \begin{pmatrix} e_{j_1}^T \\ \vdots \\ e_{j_k}^T \end{pmatrix} U \begin{pmatrix} e_{\tau_1} & \cdots & e_{\tau_k} \end{pmatrix} \tag{3.3.31}$$

と定義する．ただし e_i は第 i 成分のみが 1 でそれ以外は 0 のベクトルである．Cuppen の DC 法と同様に, 小問題 T_i, (i=1,2,3) の結果を基に有理関数 (3.3.30) を決定し, 有理方程式 $f(\lambda) = 0$ を解くことにより T の 固有値を求めることができる．DC 法と比較して更に小サイズの問題に分解するという意味で, 本手法を highly divided and conquer 法 (以下 HDC と呼ぶ) として提案する．

本手法は secular 方程式の極の次数が, DC 法とは異なるため係数の計算方法や根の存在範囲も異なってくる．以下その性質について議論する．

d_j が全て異なる場合

とくに $d_1, \ldots, d_n$ が全て異なる場合には, 式 (3.3.30) が d_j を一次の極とする部分分数に分けることができるので, 次式の様に書くことができる．

$$f(\lambda) = 1 - \sum_{j=1}^{n} \left(\sum_{\tau=1}^{r} (U_{j;\tau})^2 + \sum_{k\neq j} \frac{(\det U_{j,k;1,2})^2}{(d_k - d_j)} \right) \frac{1}{\lambda - d_j} \tag{3.3.32}$$

$$= 1 - \sum_{j=1}^{n} \frac{\omega_j}{\lambda - d_j} \tag{3.3.33}$$

$$\text{ただし} \quad \omega_j = \sum_{\tau=1}^{2} (U_{j;\tau})^2 + \sum_{k\neq j} \frac{(\det U_{j,k;1,2})^2}{(d_k - d_j)} \tag{3.3.34}$$

上式 (3.3.33) は式 (3.3.16) と同形式であるので, あらかじめ ω_j を計算しておけば根探索が同様にできる．分割数 3 で ω_j の計算の手間は第一項目が $O(1)$ で第二項目が $O(n)$ である．それ以上の分割数の場合を考えたとする, 分割数 4 の場合には式 (3.3.30) において $\sum^{r(=2)}$ を $\sum^{r(=3)}$ と置き換えればよい．極が全て分離しているとして, (3.3.33) のように整理すると計算量は $O(n^2)$ となり $f(\lambda)$ の計算を上回ってしまう ($f(\lambda)$ の計算量は $O(n)$)．したがって, 係数 ω_i の計算量が $f(\lambda)$ の計算量を越えない最大の分割数という意味で, 分割数 3 が最適になることが分かる．

分割数 2 のときの secular 方程式と同様に, 分割数 3 の場合も Courant-Weyl の原理 [40] から根の存在範囲が与えられる．$d_1 \le \ldots \le d_n$ と昇順に並べられているとすると

$$d_k \le \lambda_k, \qquad 1 \le k \le n \tag{3.3.35}$$

$$\lambda_k \le d_{k+2}, \qquad 1 \le k \le n-2 \tag{3.3.36}$$

$$d_{n+1-k} + \lambda_{n+1-j}(VV^T) \ge \lambda_{n+2-k-j}, \qquad 1 \le k, j \le n,\ k+j \le n+1 \tag{3.3.37}$$

が成立する．第3式の $\lambda_i(A)$ の表記は A の第 i 番目の固有値を表す．今 V の構成法から $\lambda(VV^T)$ は容易に求めることができる．$\lambda_{1\ldots n-2}(VV^T)=0$, $\lambda_{n-1}(VV^T)=2\min\{\rho_1,\rho_2\}$, $\lambda_n(VV^T)=2\max\{\rho_1,\rho_2\}$ を利用すると，

$$\lambda_k \le d_k + 2\max\{\rho_1,\rho_2\}, \qquad 1\le k\le n \tag{3.3.38}$$

$$\lambda_k \le d_{k+1} + 2\min\{\rho_1,\rho_2\}, \qquad 1\le k\le n-1 \tag{3.3.39}$$

が導出される．ただし，$\mathrm{size}T_2>1$, つまり VV^T は孤立した2つの 2×2 行列からなるものとする．$\mathrm{size}T_2=1$ の場合は，

$$\left\{\begin{array}{l} 2\max\{\rho_1,\rho_2\} \\ 2\min\{\rho_1,\rho_2\} \end{array}\right. \Longrightarrow \left\{\begin{array}{l} \alpha+\beta \\ \alpha-\beta \end{array}\right. , \qquad \alpha=\rho_1+\rho_2,\ \beta=\sqrt{\rho_1^2-\rho_1\rho_2+\rho_2^2} \tag{3.3.40}$$

とそれぞれ置き換えるものとする．

The deflation step

2分割の DC 法の場合と同様に，deflation 操作を行なうことができる．DC 法の場合，deflation を引き起こしていた要因として2つが存在した．まず第一に，$z_i=0$ となる成分が存在し $zz^Te_j=0$ を成立させることにあった．3分割の場合も，$U_{j;1}=U_{j;2}=0$ のとき $UU^Te_j=0$ となる．またそれ以外に，$\omega_i=0$ となるときに d_i が縮退した極として扱われるので deflation 操作 の対象になる．

またもう一つの要因として，d_j の重複による deflation がある．$d_i=d_{i+1}=\cdots=d_{i+p-1}$ とするとき，$p\ge 3$ であれば、ある適当な相似変換によって，$i+2\sim i+p-1$ 部分を 0 にできる(この事実は任意の行列の QR 変換によって保証される)．また，$U_{i\cdots i+p-1;1,2}$ のランクが1になる場合も考えられるが，その場合には $U_{i\cdots i+p-1;1}$ と $U_{i\cdots i+p-1;2}$ が定数倍の違いしかないということを意味しており DC 法で行った回転操作によって $i+1$ 部分も含めて $i+p-1$ まで 0 とできる．

secular 方程式の表現を簡潔にするため，さらに一般化して，d_k が一致するものとその重複度を含めて次のようにグループ化する．

$$\underbrace{(d_{i_1},\ldots,d_{i_1+p_1-1})}_{p_1\text{個}},\underbrace{(d_{i_2},\ldots,d_{i_2+p_2-1})}_{p_2\text{個}},\ldots,\underbrace{(d_{i_l},\ldots,d_{i_l+p_l-1})}_{p_l\text{個}} \tag{3.3.41}$$

つまり d_{i_j} は重複度 p_j である，異なる d_k は高々 l 個だけであることを意味する．このとき，d_i の重複度が2以上のものを含むときの secular 方程式の表現は次のように与えられる．

$$f(\lambda)=1-\sum_{j=1}^{l}\frac{\omega_j}{\lambda-d_{i_j}}+\sum_{j=1}^{l}\frac{\eta_j}{(\lambda-d_{i_j})^2} \tag{3.3.42}$$

$$\text{ただし } \omega_j=\sum_{i=i_j}^{i_j+p_j-1}\left(\sum_{\tau=1}^{2}(U_{i;\tau})^2+\sum_{d_k\neq d_i}\frac{(\det U_{i,k;1,2})^2}{(d_k-d_i)}\right) \tag{3.3.43}$$

$$\eta_j=\sum_{i_j\le i<k\le i_j+p_j-1}(\det U_{i,k;1,2})^2 \tag{3.3.44}$$

η_j の構成からも明らかな様に, $p_j = 1$ の場合は $\eta_j = 0$ である. したがって 式 (3.3.42) は式 (3.3.33) も含んでいる.

上式の 各 η_j と ω_j の定義を元にすると, それぞれ $O({p_j}^2)$ と $O(p_j(n-p_j))$ の計算量を要すが, 実際には U の構成方法からゼロ成分が多く存在するため $U_{i;1} = U_{i;2} = 0$ または $U_{k;1} = U_{k;2} = 0$ となる可能性が高く, $\det U_{i,k;1,2} = 0$ となることが殆んどである. また, deflation 操作によって非ゼロ成分も殆んどの項が 0 になるため実質の計算量はかなり少なくなる.

また, η の構成方法から, η_j が 0 か否かで $U_{i_j \cdots i_j+p_j-1;1,2}$ のランクを得ることができる. つまり $\eta_j \neq 0$ が成り立てば ランクは 2 であるし, $\eta_j = 0$ になれば ランクは 1 以下となるからである. ランク 0 は $U_{i_j \cdots i_j+p_j-1;1,2} = 0$ を意味するので, その判別は容易である. ランクが 2 になる場合には ω_j の値によって判別を行うことになる. したがって, これら $U_{i_j \cdots i_j+p_j-1;1,2}$ のランク判別により, 重複根 d_j が幾つ T の固有値になるかを決定することができる。

3.3.5 HDC 法に現われる secular 方程式の例

本節では, T の具体的な分割方法を示すとともに, それに対応する secular 方程式の根ならびに deflation 操作の有無について例示する. なお, 行列 0 要素についてはアルゴリズムの説明に特に必要な $\rho_{\{1,2\}}$ 等の部分を除いて空白としている.

例. 1

$$\begin{pmatrix} 2 & 1 & & \\ 1 & 2 & 1 & \\ & 1 & 2 & 1 \\ & & 1 & 4 \end{pmatrix} = \begin{pmatrix} 1 & & & \\ & 1 & 1 & \\ & 1 & 1 & \\ & & & 3 \end{pmatrix} + \begin{pmatrix} 1 & \\ 1 & \\ & 1 \\ & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & & \\ & & 1 & 1 \end{pmatrix} \tag{3.3.45}$$

この問題の場合 D, Q, U はそれぞれ次のようになる.

$$D = \mathrm{diag}(0,1,2,3), \quad Q = \begin{pmatrix} & 1 & & \\ \frac{1}{\sqrt{2}} & & \frac{1}{\sqrt{2}} & \\ \frac{-1}{\sqrt{2}} & & \frac{1}{\sqrt{2}} & \\ & & & 1 \end{pmatrix}, \quad U = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} & \\ 1 & & \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \\ & 1 & \end{pmatrix}$$

これより, 式 (3.3.33) を用いて $f(\lambda)$ ならびに, この行列の特性方程式 $g(\lambda)$ が計算できる.

$$f(\lambda) = 1 - \frac{3/2}{\lambda - 1} - \frac{1/2}{\lambda - 2} - \frac{13/6}{\lambda} - \frac{-1/6}{\lambda - 3} \tag{3.3.46}$$

$$g(\lambda) = \lambda^4 - 10\lambda^3 + 33\lambda^2 - 40\lambda + 13 \tag{3.3.47}$$

また, 根 λ_i の存在区間は順に $[0,2]$, $[1,3]$, $[2,5]$, $[3,5]$ である. $f(\lambda)$ と $g(\lambda)$ を同時にプロットすると図 3.28 のようになる。本例は極が全て異なる場合を示したものである. 2 分割の DC 法の secular 方程式の振る舞いと類似してはいるが, $\omega_i < 0$ となるため根が極の右側に必ずしも存在していない.

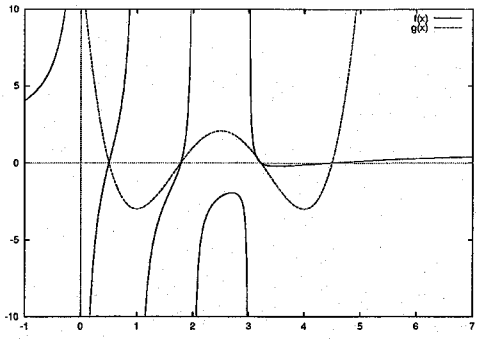


図 3.28: 例.1 における $f(\lambda)$ と $g(\lambda)$

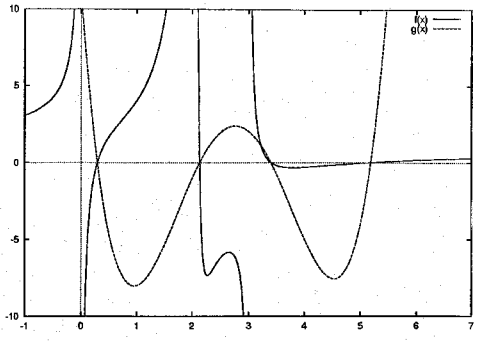


図 3.29: 例.2 における $f(\lambda)$ と $g(\lambda)$

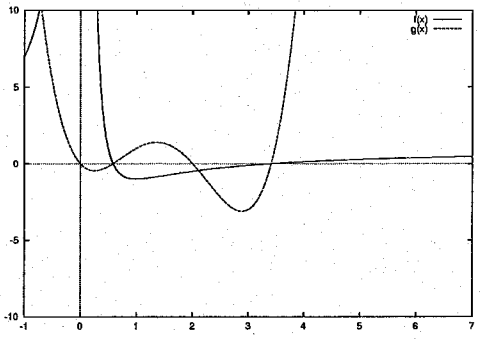


図 3.30: 例.3 における $f(\lambda)$ と $g(\lambda)$

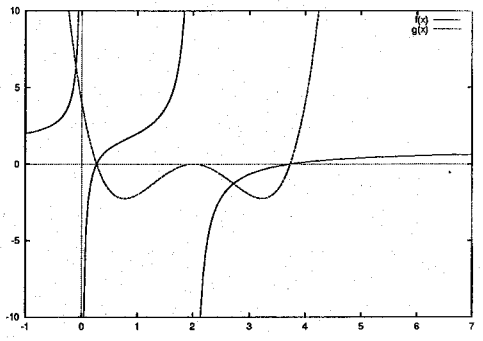


図 3.31: 例.4 における $f(\lambda)$ と $g(\lambda)$

例. 2

$$\begin{pmatrix} 1 & 1 & & \\ 1 & 2 & 1 & \\ & 1 & 4 & 1 \\ & & 1 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 1 & & \\ 1 & 1 & & \\ & & 2 & \\ & & & 3 \end{pmatrix} + \begin{pmatrix} & \\ 1 & \\ 1 & 1 \\ & 1 \end{pmatrix} \begin{pmatrix} & 1 & 1 & \\ & & 1 & 1 \end{pmatrix} \tag{3.3.48}$$

D, Q, U はそれぞれ

$$D = \mathrm{diag}(0,2,2,3),\quad Q = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & & \\ \frac{-1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & & \\ & & 1 & \\ & & & 1 \end{pmatrix},\quad U = \begin{pmatrix} \frac{-1}{\sqrt{2}} & \\ \frac{1}{\sqrt{2}} & \\ 1 & 1 \\ & 1 \end{pmatrix}$$

となる. 重複根 $d_i = 2$ を持つので, 式 (3.3.42) を用いて $f(\lambda)$ ならびに, この行列の特性方程式 $g(\lambda)$ が計算できる.

$$f(\lambda) = 1 - \frac{15/4}{\lambda - 2} - \frac{11/12}{\lambda} - \frac{-2/3}{\lambda - 3} + \frac{1/2}{(\lambda - 2)^2} \tag{3.3.49}$$

$$g(\lambda) = \lambda^4 - 11\lambda^3 + 39\lambda^2 - 48\lambda + 11 \tag{3.3.50}$$

根 λ_i の存在範囲は順に $[0,2]$, $[2,3]$, $[2,4]$, $[3,6]$ である.

$1/(\lambda-2)^2$ の 係数が 0 でないので, $d_i=2$ での deflation が起こらず 2 は T の固有値に含まれない. 実際に特性方程式 $g(\lambda)$ と $f(\lambda)$ を同時にプロットしたものを比較するそれが確認できる (図 3.29).

例. 3

$$\begin{pmatrix} 1 & 1 & & \\ 1 & 2 & 1 & \\ & 1 & 2 & 1 \\ & & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & & \\ 1 & 1 & & \\ & & 0 & \\ & & & 0 \end{pmatrix} + \begin{pmatrix} & & & \\ 1 & & & \\ 1 & 1 & & \\ & 1 & & \end{pmatrix} \begin{pmatrix} & 1 & 1 & \\ & & 1 & 1 \end{pmatrix} \tag{3.3.51}$$

D, Q, U はそれぞれ

$$D = \mathrm{diag}(0,0,0,2),\quad Q = \begin{pmatrix} \frac{1}{\sqrt{2}} & & & \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} & & & \frac{1}{\sqrt{2}} \\ & 1 & & \\ & & 1 & \end{pmatrix},\quad U = \begin{pmatrix} \frac{-1}{\sqrt{2}} & & \\ 1 & 1 & \\ & 1 & \\ \frac{1}{\sqrt{2}} & & \end{pmatrix}$$

式 (3.3.42) を用いて $f(\lambda)$ ならびに, この行列の特性方程式 $g(\lambda)$ が計算できる.

$$f(\lambda) = 1 - \frac{0}{\lambda-2} - \frac{4}{\lambda} + \frac{2}{\lambda^2} \tag{3.3.52}$$

$$g(\lambda) = \lambda^4 - 6\lambda^3 + 10\lambda^2 - 4\lambda \tag{3.3.53}$$

となる。根 λ_i の存在範囲は順に $[0,1]$, $[0,1]$, $[0,3]$, $[2,5]$ である. $d_i=0$ が 3 重の根であるので deflation を起こすが $\eta \neq 0$ であるので deflation を起こすのは 1 つの 0 だけである。さらに $d_i=2$ の係数が 0 であるのでこれもまた deflation の対象になる (図 3.30)。

例. 4

$$\begin{pmatrix} 1 & 1 & & \\ 1 & 2 & 1 & \\ & 1 & 3 & 0 \\ & & 0 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 1 & & \\ 1 & 1 & & \\ & & 2 & \\ & & & 2 \end{pmatrix} + \begin{pmatrix} & & & \\ 1 & & & \\ 1 & 0 & & \\ & 0 & & \end{pmatrix} \begin{pmatrix} & 1 & 1 & \\ & & 0 & 0 \end{pmatrix} \tag{3.3.54}$$

D, Q, U はそれぞれ

$$D = \mathrm{diag}(0,2,2,2),\quad Q = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & & \\ \frac{-1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & & \\ & & 1 & \\ & & & 1 \end{pmatrix},\quad U = \begin{pmatrix} \frac{-1}{\sqrt{2}} & \\ \frac{1}{\sqrt{2}} & \\ 1 & 0 \\ & 0 \end{pmatrix}$$

式 (3.3.42) を用いて $f(\lambda)$ ならびに, この行列の特性方程式 $g(\lambda)$ が計算できる.

$$f(\lambda) = 1 - \frac{1/2}{\lambda} - \frac{3/2}{\lambda-2} + \frac{0}{(\lambda-2)^2} \tag{3.3.55}$$

$$g(\lambda) = \lambda^4 - 8\lambda^3 + 21\lambda^2 - 21\lambda + 68 \tag{3.3.56}$$

となる。根 λ_i の存在範囲は順に $[0,2]$, $[2,2](=2)$, $[2,2](=2)$, $[2,4]$ である. $d_i=2$ が 3 重の根であるので deflation を起こす. $\eta=0$ であるので deflation を起こすのは 2 つの $d_i=2$ である (図 3.31). これは, 根の存在区間にある $\lambda_i=2$ が 2 つあることと一致している.

3.3.6 HDC 法の根探索のための初期区間選定

根の存在区間は式 (3.3.35) から (3.3.39) の 5 式によって特定される. k 番目の根 λ_k の存在区間の下限を LB_k, 上限を UB_k とおく.

$$\mathrm{LB}_k \leq \lambda_k \leq \mathrm{UB}_k \tag{3.3.57}$$

$n = 6$ の場合について, 根の存在区間を示した例を図 3.32に示す. ただし, $\{d_i\}_{i=1,\ldots,6} = \{0,\ 1.0,\ 3.0,\ 3.5,\ 4.0,\ 5.5\}$, $\rho_1 = 0.25, \rho_2 = 1.0$ である. 図 3.32から分かるように, DC 法とは異なり, 大部分の根の存在区間に重なりが生じている.

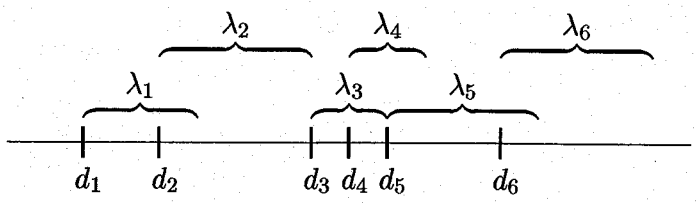


図 3.32: 根 λ の存在区間

λ_k の下限 LB_k は d_k に固定されるが, $d_{k+1} - d_k < 2\max\{\rho 1, \rho 2\}$ が成立するとき UB_k は d_{k+1} を越えてしまうので, すぐ隣の固有値の区間と重なりを起こすこととなる. secular 方程式 $f(\lambda) = 0$ を解くためには, なるべく区間内に根が 1 つだけに限定されていた方がよい. そこで区間をより限定するために, $d_{k+1} < \mathrm{UB}_k$ となるときには $\lambda = d_{k+1}$ における小行列式のストゥルム列の符号数を数えあげ, (LB_k, d_{k+1}) もしくは (d_{k+1}, UB_k) のいずれに期待する根が存在するかを特定すればよい. 同様に LB_{k+1} も適切な区間に設定し直す.

しかしながら特定した区間の中に 2 つの根が存在し, 区間が一致してしまうことが起こり得る. このような場合には, 小さい方の根は区間の下限に近い極近傍部分に存在し, 大きい方は上限に近いもう一方の極近傍部分に存在していることが経験的に理解できる. 2 つの根の初期値を両極の近傍値にうまく設定し Newton 法を用いることで 2 つの根を分けて求めることができる.

以上の議論をまとめ, DC 法の節で議論した並列処理を HDC 法にも適用を考えると, 計算ステップは次 (図 3.33) の様にまとめられる.

1. $T = \begin{pmatrix} T_1 & & \\ & T_2 & \\ & & T_3 \end{pmatrix} + VV^T$ に 分ける.

2. 3 つに分けた小問題 T_1, T_2, T_3 の固有値及び固有ベクトルを並列に計算する.
   各問題の固有値計算にも複数のプロセッサを使った並列計算を行なう.

3. 固有値 d_k のソーティングを行なうとともに, 対応する固有ベクトル q_k の並び替えも行う.
   重複固有値が存在する場合には, $(d_{i_1}, ..., d_{i_1+p_1-1}), (d_{i_2}, ..., d_{i_2+p_2-1}), ...$ の様にグループ化を施す.

4. $U = Q^T V$ を計算.

5. 次式で表す様に, secular 方程式の係数 ω_j, η_j を計算する.

$$f(\lambda) = 1 - \sum_{j=1}^{l} \frac{\omega_j}{\lambda - d_j} + \sum_{j=1}^{l} \frac{\eta_i}{(\lambda - d_i)^2}$$

$$\omega_j = \sum_{i=i_j}^{i_j+p_j-1} \left(\sum_{\tau=1}^{2} (U_{j;\tau})^2 + \sum_{d_k \neq d_j} \frac{(\det U_{j,k;1,2})^2}{(d_k - d_j)} \right)$$

$$\eta_i = \sum_{i_j \leq i < k \leq i_j+p_j-1} (\det U_{j,k;1,2})^2$$

6. $|\omega_j| < \varepsilon$, $|\eta_j| < \varepsilon$ の有無により deflation を行い, 探索すべき固有値の数を削減する.

7. 存在区間 $[\mathrm{LB}_k, \mathrm{UB}_k]$ を特定する。

$$\mathrm{LB}_k = \quad d_k$$

$$\mathrm{UB}_k = \begin{cases} \min\{d_{k+2}, d_k + 2\max\{\rho_1, \rho_2\}, d_{k+1} + 2\min\{\rho_1, \rho_2\}\} & 1 \leq k \leq n-2 \\ \min\{d_k + 2\max\{\rho_1, \rho_2\}, d_{k+1} + 2\min\{\rho_1, \rho_2\}\} & k = n-1 \\ d_k + 2\max\{\rho_1, \rho_2\} & k = n \end{cases}$$

8. $[\mathrm{LB}_k, \mathrm{UB}_k] \cap [\mathrm{LB}_{k+1}, \mathrm{UB}_{k+1}] \neq \emptyset$ の場合, $\lambda = d_{k+1}$ においてストゥルム列の符合数え上げによって何番目の固有値が区間に含まれているかを特定する. その情報をもとに厳密な初期区間の上限 $\widetilde{\mathrm{UB}}_k$ と下限 $\widetilde{\mathrm{LB}}_{k+1}$ の特定を行なう.

9. 初期区間 $[\widetilde{\mathrm{LB}}_k, \widetilde{\mathrm{UB}}_k]$ をもとに, Newton 法などの手法を用いて $f(\lambda) = 0$ の根を計算する.

図 3.33: 並列 HDC 法の計算のためのアルゴリズム

3.3.7 DC 法ならびに HDC 法の計算実験結果と性能評価

テスト行列として, 全ての固有値の分布が分かっている 3 重対角行列を選んだ.

$$A = \begin{pmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & 1 & -2 & 1 & \\ & & \ddots & \ddots & \ddots \\ & & & 1 & -2 \end{pmatrix} \tag{3.3.58}$$

この行列は 差分における Δ(ラプラシアン) の行列表現であり, この行列の固有値は

$$\lambda_i = -2\left(1 - \cos\frac{j\pi}{n+1}\right), \quad (i = 1, 2, \ldots, n) \tag{3.3.59}$$

と解析的に計算できる.

この行列に対して並列計算機 ADENART256 を用いて固有値の計算を 2 分法, DC 法, HDC 法の 3 通りで行い, 行列のサイズを変化させて比較を行った. 2 分法は EISPACK[37] で用いられている関数`bisect` を並列化したものである. なお, DC 法ならびに HDC 法では問題を再帰的に小さい問題 (例えば最小の 1×1 行列) まで分割することが可能であるが, 今回は 1 レベルの分割にとどめ効果を見てみた. また, 下位問題への分割は重複根がでないように設定している. 2 分法, DC/HDC 法での Newton 法で用いた反復法の打ち切りは, 次の条件で行った.

$$\frac{|x_{k+1} - x_k|}{|x_{k+1}| + |x_k|} < \epsilon(\approx 10^{-15}) \tag{3.3.60}$$

数値実験の結果は表 3.1にあるようになる. 括弧内の数字は 2 分法を基準にした時の比を表す. なお, DC 法, HDC 法ともに下位問題での固有値求解には 2 分法と逆反復法を用いて計算しており, 表中の DC,HDC の計算時間には下位問題での求解に要した時間も含まれている. 行列のサイズに依存する部分があるが, DC 法で 10~20%, HDC 法で 20~30% の高速化が見られている.

次に計算精度についての検討を行なってみる. 固有値の真値が分かっているので, 表 3.2において適当な部分の固有値を選択して真値と比較をしてみた. 行列のサイズは $N = 1024$, 下位問題の分割は $(N_1, N_2, N_3) = (361, 286, 377)$ である.

結果からも分かるように絶対値が小さい方の固有値の精度があまりよくないことが分かる。 DC 法の下位に位置する 2 分法でこの誤差が起こっているので, DC 法でその誤差を含むということになる. そのために DC/HDC 法で精度の改善を行なうのは難しいと考えられる. 適当な精度までの近似解を探索し, 2 分法などによって精度を上げるハイブリッドな手法が効果的である.

また、HDC 法の λ_{1023} と λ_{1024} が入れ替わっていたり、HDC での λ_{615} が真値と著しく異なっているのがある。前者は初期値の選択によって起こったものであり偶然異なる根に収束したものと思われる。後者は $\omega_{615}, \omega_{616}$ の計算過程で, 生じる誤差のために厳密に 0 と判断できないことによって生じるものである. 実際 ω_{616}= -2.802966192483325E-14 となっており、マシンイプシロン $\varepsilon(10^{-15})$ では 0 と判断できていなかった. この誤差を回避するためには, 計算過程で登場する適当な正数 (本論文では $|T|$ を採用した) でスケーリングして $|\omega| < |T|\varepsilon$ が成立する場合に 0

とみなすことで回避できる．本例の場合，この回避方法で deflation の対象として Newton 法の探索から取り除かれ，十分高い精度の解 $\lambda_{615} = -1.381966011250016$(真値 $-$ 1.381966011250106) を得ることができる．

DC/HDC 法が secular 方程式の求解を行う性質上固有値の精度をあげるには secular 方程式を精度良く表現しかつ精度良く解くことが要請される．つまり，下位のルーチンの精度をあげること，係数 ω の計算を精度良く行うことである．また，本稿でも精度面に関する改善案を示しているように根探索における 0 判定方法や初期値などの選び方を改良する必要があることがわかる．

表 3.1: ADENART256 上での固有値計算時間 (単位: ミリ秒): 各アルゴリズムに対して，行列サイズ (1024,1536,2048,2540) での実行時間と 2 分法の計算時間を 100% としたときの比を括弧内に示した．

	1024		1536		2048		2540	
2 分法	1550	(100%)	3388	(100%)	5916	(100%)	9123	(100%)
DC	1143	(73%)	2551	(75%)	4902	(82%)	8082	(88%)
HDC	1255	(80%)	2437	(71%)	4107	(69%)	6652	(72%)

表 3.2: 2 分法, DC 法, HDC 法で計算された固有値の精度比較

固有値	2 分法	DC 法	HDC 法	真値
λ_1	-3.9999906059758	-3.9999906059758	-3.9999906059758	-3.9999906059758
λ_2	-3.9999624239914	-3.9999624239914	-3.9999624239914	-3.9999624239914
λ_3	-3.9999154543117	-3.9999154543117	-3.9999154543117	-3.9999154543117
λ_4	-3.9998496973778	-3.9998496973778	-3.9998496973778	-3.9998496973778
λ_5	-3.9997651538074	-3.9997651538074	-3.9997651538074	-3.9997651538074
λ_{509}	-2.0214543676085	-2.0214543676085	-2.0214543676085	-2.0214543676085
λ_{510}	-2.0153246922526	-2.0153246922526	-2.0153246922526	-2.0153246922526
λ_{511}	-2.0091948729362	-2.0091948729362	-2.0091948729362	-2.0091948729362
λ_{512}	-2.0030649672428	-2.0030649672428	-2.0030649672428	-2.0030649672428
λ_{513}	-1.9969350327572	-1.9969350327572	-1.9969350327572	-1.9969350327572
λ_{614}	-1.387798821454504	-1.387798821454504	-1.387798821454541	-1.387798821454504
λ_{615}	-1.381966011250105	-1.381966011250105	-1.381957659670318	-1.381966011250106
λ_{616}	-1.376139006871953	-1.376139006871953	-1.376139006872044	-1.376139006871953
λ_{617}	-1.370317863059067	-1.370317863059067	-1.370317863058977	-1.370317863059068
λ_{618}	-1.364502634495414	-1.364502634495414	-1.364502634495481	-1.364502634495414
λ_{1020}	-2.3484619263706D-04	-2.3484619263751D-04	-2.3484619264013D-04	-2.3484619263714D-04
λ_{1021}	-1.5030262224816D-04	-1.5030262224816D-04	-1.5030262224593D-04	-1.5030262224802D-04
λ_{1022}	-8.4545688311982D-05	-8.4545688311144D-05	-8.4545688311144D-05	-8.4545688312065D-05
λ_{1023}	-3.7576008551282D-05	-3.7576008551282D-05	-9.3940241956552D-06	-3.7576008551143D-05
λ_{1024}	-9.3940241997770D-06	-9.3940242026780D-06	-3.7576008557614D-05	-9.3940241996382D-06

第 4 章

応用問題への適用

本章では応用プログラムとして位置づけ可能な大規模な問題に対する並列化の議論を行う．まず並列計算機の性能測定に利用するNASパラレルベンチーマーク1版を取り上げ，その並列化を並列処理様式ADEPSモデルをもとに行った結果を報告する．次に，粒子プラズマ問題を取り上げてその並列化について報告する．ここで取り上げる粒子プラズマ問題は非常に多くの粒子を取り扱わなくてはならないため大容量の記憶装置そして演算能力が必要とされる問題であり、並列処理が必須となる問題の一例である．

4.1 NASパラレルベンチマーク

NASパラレルベンチマーク (以下NPBと呼ぶ)[4] は，測定対象である計算機上でこれらプログラム群の実行に要した経過時間を計測し，基準となる計算機での経過時間との相対比性能でもって性能を評価するものである．

NPBは8つのカーネルプログラムと3つの流体アプリケーションのコア部分から構成されるテストプログラム群である．これらは，並列計算機の様々な特性を調査するための特化されており調査項目毎に利用するプログラムが決まっている．特に5つのカーネルプログラムは科学技術計算において要求される基本的な幾つかの項目を含んでいる．また，3つの流体アプリケーションは典型的な流体プログラムの中に含まれる異なる差分スキームならびに線形方程式求解法を用いている．

NASパラレルベンチマークには，1991年に制定された第一版 (以下NPB1)[4] と，その後1995年に制定された第二版 (以下NPB2)[5] とが存在する．NPB1は “pencil and paper” のアプローチによるベンチマークと呼ばれ，書面上で与えられたアルゴリズムをベンチマークが規定するルールの範囲でプログラム作成できる自由度があった．一方NPB2では，逐次プログラムとMPIを用いて並列化された二つのコードが用意されている．そして，利用者はこれらに対して，5 % 以内のプログラムを実行にするための最低限の書き換えが許されている．そういった意味でNPB1とNPB2は大きく異なり，マシンの性能とコンパイラの最適化をより公正に比較できるように改良が加えられている．NPBは現在も進化の途中であり，最新のものはNPB2.3が公開されている．NPB2.3にはカーネルプログラムやCFDアプリケーションの実装に幾つかの修正が加わって

いる.

研究 [20] では, NPB1 の精神に基づき指定されたアルゴリズムを独自の手法により並列化し, 並列計算機上での実装並びに性能を測定した. ちなみに, 実験ならびに測定に用いた計算機は ADENART/-256 であり, 京都大学と松下電器産業とが共同で開発した論理ピーク性能 2.2GFLOPS(34MHz 動作時) の並列計算機である. ADENART/256 は 256 台のスカラプロセッサを搭載する MIMD 分散メモリ型並列計算機である. 個々のプロセッサはハイパークロスネット (以下 HX ネットと呼ぶ) によって結合されている. HX ネットの総データ転送性能は 2.2G バイト / 秒に達する.

4.1.1 カーネル EP

カーネル EP は, モンテカルロ法に基づくある種の統計量の収集を行うプログラムである. 乱数の生成ならびに統計量の採取は個々のプロセッサ毎に独立して実行し, 最終的に全プロセッサで総和をとることによってプログラムが完結する (図 4.1).

そういった意味で, カーネル EP は, 'Embarrassingly Parallel' つまり並列計算を行う際に通信も必要なく自明な並列性を有するプログラムである. そのため, カーネル EP は並列システムが装備する浮動小数点演算機の演算性能の上限を測定する役割をなす.

カーネル EP の並列化は次のように行った.

1. NPB1 版レポートのアルゴリズム作成指針にもあるように, 始めに個々のプロセッサに乱数の初期種を割り当てる.

2. その後, 割り当てられた初期種だけを元に各々のプロセッサが計算すべきサンプリング点の個数だけ計算を進行する.

4.1.2 カーネル MG

カーネル MG は, 3 次元ポアソン方程式をマルチグリッド法によって解くプログラムである. MG で採用されているマルチグリッド法はもっとも単純な V- サイクル手法である. V- サイクルでは, 初期のグリッドの解像度を $1/2, 1/4, \ldots, 1/2^n$ と粗くしていきながら求解し, それを解像度の高い問題に対してスムージングを行うことによって近似解とする手法である (図 4.2).

カーネル MG は, 近接するグリッドの平滑 (平均) 化を行うために近接グリッドへのアクセス速度を測定するプログラムとなっている. 領域分割によって, 空間領域を各プロセッサ割り付けた場合には直ぐ隣のプロセッサへのアクセス, つまり近距離通信の性能測定を行うベンチマークプログラムと言える. MG の並列化には, ADEPS に基づく交互方向のアクセスによるモデルによって並列化を行った. MG のコア部分は次の様に表すことができる.

$$A^{n+1}(\mathrm{P}) = \alpha_0 A^n(\mathrm{P}) + \alpha_1 A^n(\mathrm{P}-1) + \alpha_2 A^n(\mathrm{P}-2) + \alpha_3 A^n(\mathrm{P}-3) \qquad (4.1.1)$$

ここで, P はある格子点上の座標を表し, $\mathrm{P}-1$ は P から x, y, z の変位の総和が 1 となる全ての点を意味する. この式は次式の様に x-, y-, z- 方向の演算に分解される.

parallel do
  $k \leftarrow 0$
  while ($k < N$)
    generate random-pair (x_j, y_j) and set $t_j = x_j^2 + y_j^2$
    if $t_j \leq 1$ **then**
      $k \leftarrow k+1$, $X_k := x_j\sqrt{(-2\log t_j)/t_j}$ and $Y_k := y_j\sqrt{(-2\log t_j)/t_j}$
      $Q_l \leftarrow Q_l + 1$, where l holds $l \leq \max\{|X_k|, |Y_k|\} \leq l+1$
    endif
  end
enddo
gather Q_l $l = 1, ...m$
print Q_l $l = 1, ..., m$

図 4.1: カーネル EP のアルゴリズム

solve $z_k = M^k r_k$:
  if $(k > 1)$ **then**
    $r_{k-1} = Pr_k$ (restrict residual)
    $z_{k-1} = M^{k-1} r_{k-1}$ (recursive solve)
    $z_k = Qz_{k-1}$ (prolongate)
    $r_k = r_k - Az_k$ (evaluate residual)
    $z_k = z_k + Sr_k$ (apply smoother)
  else
    $z_1 = Sr_1$ (apply smoother)
  endif

図 4.2: カーネル MG における V- サイクル操作

$$A^{n+1}(\mathrm{P}) = \Big\{(\alpha_0 A^n + \alpha_1 A^n_{(0,0,\pm1)}) + (\alpha_1 B + \alpha_2 B_{(0,0,\pm1)}) + (\alpha_1 C + \alpha_2 C_{(0,0,\pm1)}) + (\alpha_2 D + \alpha_3 D_{(0,0,\pm1)})\Big\}(\mathrm{P}) \tag{4.1.2}$$

$$B(\mathrm{P}) = (A^n_{(\pm1,0,0)})(\mathrm{P}) \tag{4.1.3}$$

$$C(\mathrm{P}) = (A^n_{(0,\pm1,0)})(\mathrm{P}) \tag{4.1.4}$$

$$D(\mathrm{P}) = (B^n_{(0,\pm1,0)})(\mathrm{P}) \tag{4.1.5}$$

ここで, 下添字 (i,j,k) は点 P からの変位を表す. 図 4.3にコア部分計算に必要なデータへのアクセス. 図 4.4に交互方向に分解した場合のアクセスの遷移過程を表す.

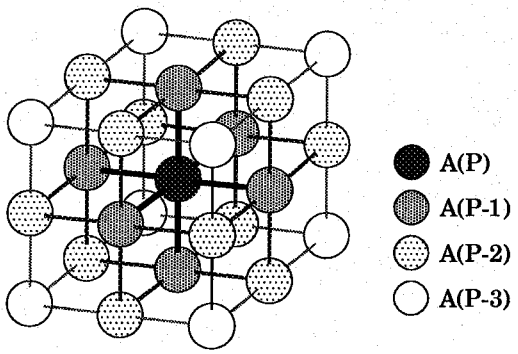


図 4.3: マルチグリッド法における $A(P)$ のコアアクセス

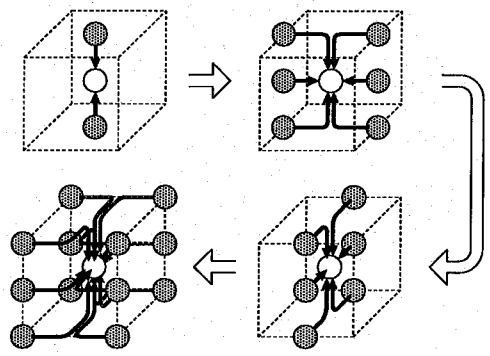


図 4.4: 効率的な交互方向の分解とアクセス遷移過程

式 (4.1.1) の各項を順に計算した場合には距離 0 の計算に 1 ステップ, 距離 1,2,3 の計算にそれぞれ 3,5,3 ステップ, つまり合計 12 ステップの処理が必要である. 式 (4.1.2) から (4.1.5) を用いた場合には, 中間変数 B, C, D を導入することで 7 ステップで計算が完了させることができる (ただし距離 0 と z-index が 1 ずれたものとの和を 1 ステップで計算するとして). 特に, コア部分の係数に $\alpha_0 = a, \alpha_1 = ab, \alpha_2 = ab^2, \alpha_3 = ab^3$ なる性質がある場合には, 次のような更なる変形が可能である.

$$A^{n+\frac{1}{3}}(\mathrm{P}) = (A + b(A_{(0,0,\pm1)}))^n(\mathrm{P}) \tag{4.1.6}$$

$$A^{n+\frac{2}{3}}(\mathrm{P}) = (A + b(A_{(0,\pm1,0)}))^{n+\frac{1}{3}}(\mathrm{P}) \tag{4.1.7}$$

$$A^{n+1}(\mathrm{P}) = a(A + b(A_{(\pm1,0,0)}))^{n+\frac{2}{3}}(\mathrm{P}) \tag{4.1.8}$$

最後に示した方式では中間変数の数を 2 個に削減することができる.

4.1.3 カーネル CG

カーネル CG では, 大規模正値対称疎行列の絶対値最小固有値を求めるために逆ベキ乗法の内部で共役勾配法 (Conjugate Gradient method) を利用している. カーネル CG で用いられている

逆ベキ乗法は, NPB1 版と NPB2 版とで点列の予測の部分が異なる. NPB1 版のものを使って概略をまとめると次図 (図 4.5) の様になる. カーネル CG で用いられている共役勾配法では収束打ち切りを誤差を用いずに一定回数の反復でもって収束としている (図 4.6).

set $x = (1, 1, ..., 1)^T$
do $it = 1, niter$
  Solve the system $Az = x$ and estimate $\|r\|$ by CG method
  $\zeta = \lambda + \dfrac{1}{x^T z}$
  $x = \dfrac{z}{\|z\|}$
enddo

図 4.5: 逆ベキ乗法のアルゴリズム

set $r_0 = x, \rho_0 = \|r_0\|^2, p_1 = r_0$
do $i = 1, 25$
  $q_i = Ap_i$
  $\alpha = \rho/(p_i, q_i)$
  $z \leftarrow z + \alpha p_i$
  $\rho_i = \rho_{i-1}$
  $r_i = r_{i-1} - \alpha q_i$
  $\rho_i = \|r_i\|^2$
  $\beta = \rho_i/\rho_{i-1}$
  $p_i = r_i + \beta p_{i-1}$
enddo

図 4.6: 共役勾配法のアルゴリズム

カーネル CG で取り扱う行列の大きさは, クラス B と呼ばれる問題で約 2×10^{16} であるため, 全ての行列要素をメモリ上に格納することは困難である. また, 殆んどの要素が 0 であることが問題で規定されているので適当な構造体もしくはリストを導入することが NPB レポートでは想定されている. NPB レポートが推奨する方式をとるならば, 行列の各行に対する非ゼロ要素のリストを構成し, その要素値 (aele) と列位置 (acol) との 2 つの配列でもって配列a を表す. ADE-TRAN の記法でもってそれを表すと次の様になる.

```
a(i,j)  --->  aele(/i/,k)
              acol(/i/,k)   (=j)
```

ここで, aele(/i/,) はacol(/i/,) 列に位置する非ゼロ要素の値が格納されている. また, 共役勾配法の中では行列とベクトルとの積計算が最も計算量の多い部分である. 先に示した粗行列

のデータ構造を利用したADETRANによる実装を考えると，次に示すようなPCAST構文を用いたプログラムで簡潔に表現できる．

```
pcast i=1,n
   x(i)=x(/i/,1)
pend
pdo i=1,n
   b(/i/,1)=0.0
   do j=1,nk
      b(/i/,1)=b(/i/,1)+a(/i/,acol(/i/,j))*x(acol(/i/,j))
   enddo
pend
```

4.1.4 カーネルFT

次の様な時間発展の偏微分方程式

$$\frac{\partial u(x,t)}{\partial t} = \alpha \nabla^2 u(x,t) \tag{4.1.9}$$

はフーリエ変換によって，周波数領域で取り扱うと次の常微分方程式となる．

$$\frac{d\hat{u}(\omega;t)}{dt} = -4\alpha\pi^2|\omega|^2\hat{u}(\omega;t) \tag{4.1.10}$$

ここで$\hat{u}$はuのフーリエ変換である．この常微分方程式の解は容易に求めることができ

$$\hat{u}(\omega;t) = e^{-4\alpha\pi^2|\omega|^2}\hat{u}(\omega;0) \tag{4.1.11}$$

である．カーネルFTはフーリエ変換に離散フーリエ変換(DFT)を用いて，特に3次元の高速フーリエ変換(FFT)でもって，偏微分方程式を解くカーネル部分を取り出したものである．

FFTの一般的な性質として，特にFFTはバタフライ演算によって，2のベキのストライドで配列にアクセスするためデータを異なるプロセッサ上のメモリに分散して格納する場合には，そのアクセス速度，つまりデータ転送速度を評価するプログラムともいえる．また，3次元データのFFTは第3章でも述べた通り，x-，y-，z-方向にFFTをかける方向を切り替えながら行うADEPSでの並列化が最も自然であり，一般に必要とされるデータ再分散をADEPSモデルが吸収している．データ再分散は全プロセッサがデータ編集に参加するため，様々な距離の通信性能が影響するが，特に最も通信レイテンシが大きい遠距離プロセッサ間の通信がデータ再分散の性能を支配する．したがって，NPBではFTを遠距離プロセッサとの通信性能測定プログラムとして位置付けている．

4.1.5 CFD アプリケーションの概要

NPB に採用されている CFD アプリケーションは, 次式の様に定式化された Navier-Stokes 方程式を解くプログラムの基本部分のみを取り出したものである.

$$\frac{\partial U}{\partial \tau} = \frac{\partial E(U)}{\partial \xi} + \frac{\partial F(U)}{\partial \eta} + \frac{\partial G(U)}{\partial \zeta} + \frac{\partial T(U,U_\xi)}{\partial \xi} + \frac{\partial V(U,U_\eta)}{\partial \eta} + \frac{\partial W(U,U_\zeta)}{\partial \zeta} + H(U,U_\xi,U_\eta,U_\zeta) \tag{4.1.12}$$

where

$$U = (u^{(1)}, u^{(2)}, u^{(3)}, u^{(4)}, u^{(5)})^T$$

with boundary conditions

$$\mathcal{B}(U,U_\xi,U_\eta,U_\zeta) = U^B(\tau,\xi,\eta,\zeta), \quad (\tau,\xi,\eta,\zeta) \in D_\tau \times \partial D \tag{4.1.13}$$

and with initial conditions

$$U|_{\tau=0} = U^0(\xi,\eta,\zeta), \quad (\xi,\eta,\zeta) \in D \tag{4.1.14}$$

ここで $D \in R^3$ は問題の領域, ∂D はその境界である.

3 つのプログラムは次の問題を含む様に選定されている.

1. 独立でかつ対角非優位な, ブロックサイズが 5×5 のブロック 3 重対角システムの求解
2. 独立でかつ対角非優位な, ブロック $(5 \times 5)5$ 重対角システムの求解
3. 粗なブロック (5×5) 行列とベクトルの積
4. 粗なブロック (5×5) 上 / 下三角方程式システムの求解

ここで項目 3 は, 離散化された方程式系の右辺の値を計算するために, 全てのプログラムに含まれている. 3 つの問題の違いは, 陰解法に基づく微分方程式の時間積分に用いる 1,2,4 などの求解手法の違いである.

4.1.6 CFD アプリケーション BT

CFD アプリケーション BT は, 式 (4.1.12) の非線形項を 1 次近似し, $U^{n+1} = U^n + \Delta U^n$ の ΔU^n について以下の方程式を解くことに帰着させている.

$$\left\{ I - \Delta\tau \left[\frac{\partial(A)^n}{\partial \xi} + \frac{\partial^2(N)^n}{\partial \xi^2} + \frac{\partial(B)^n}{\partial \eta} + \frac{\partial^2(Q)^n}{\partial \eta^2} + \frac{\partial(C)^n}{\partial \zeta} + \frac{\partial^2(S)^n}{\partial \zeta^2} \right] \right\} \Delta U^n$$
$$= \Delta\tau \left[\frac{\partial(E+T)^n}{\partial \xi} + \frac{\partial(F+V)^n}{\partial \eta} + \frac{\partial(G+W)^n}{\partial \zeta} \right]$$
$$- \Delta\tau\epsilon \left[h_\xi^4 \frac{\partial^4 U^n}{\partial \xi^4} + h_\eta^4 \frac{\partial^4 U^n}{\partial \eta^4} + h_\zeta^4 \frac{\partial^4 U^n}{\partial \zeta^4} \right] + \Delta\tau H^* \tag{4.1.15}$$

方程式の左辺を次式に示す様に方向毎の作用素積に分解し, Beam-Warming アルゴリズムによって方程式を解くものである.

$$\left\{I-\Delta\tau\left[\frac{\partial(A)^n}{\partial\xi}+\frac{\partial^2(N)^n}{\partial\xi^2}\right]\right\}\times\left\{I-\Delta\tau\left[\frac{\partial(B)^n}{\partial\eta}+\frac{\partial^2(Q)^n}{\partial\eta^2}\right]\right\}$$
$$\times\left\{I-\Delta\tau\left[\frac{\partial(C)^n}{\partial\zeta}+\frac{\partial^2(S)^n}{\partial\zeta^2}\right]\right\}\Delta U^n=\mathrm{RHS}^n \quad (4.1.16)$$

本解法の特徴は, 各 $\xi-$, $\eta-$, $\zeta-$ 方向の陰解法部分に関して, サイズ 5×5 のブロックが並ぶ 3 重対角の方程式になっていることである. この解法は ADEPS での記述を行うと以下の様にできる. また, 各方向の 1 次元内の 3 重対角行列の方程式求解は独立なので, 効率的な並列処理が行われる.

$$\Delta U^{n+\frac{1}{3}}=(I-\Delta\tau[D_\xi(A)^n+D_\xi^2(N)^n])^{-1}\mathrm{RHS}^n \quad (4.1.17)$$
$$\Delta U^{n+\frac{2}{3}}=(I-\Delta\tau[D_\eta(B)^n+D_\eta^2(Q)^n])^{-1}\Delta U^{n+\frac{1}{3}} \quad (4.1.18)$$
$$\Delta U^{n+1}=(I-\Delta\tau[D_\zeta(C)^n+D_\zeta^2(S)^n])^{-1}\Delta U^{n+\frac{2}{3}} \quad (4.1.19)$$

4.1.7 CFD アプリケーション SP

CFD アプリケーション SP は BT で行った ΔU^n の定式化に高次の散逸項を加えたものである.

$$\left\{I-\Delta\tau\left[\frac{\partial(A)^n}{\partial\xi}+\frac{\partial^2(N)^n}{\partial\xi^2}-\epsilon h_\xi^4\frac{\partial^4(I)}{\partial\xi^4}+\frac{\partial(B)^n}{\partial\eta}+\frac{\partial^2(Q)^n}{\partial\eta^2}-\epsilon h_\eta^4\frac{\partial^4(I)}{\partial\eta^4}\right.\right.$$
$$\left.\left.+\frac{\partial(C)^n}{\partial\zeta}+\frac{\partial^2(S)^n}{\partial\zeta^2}-\epsilon h_\zeta^4\frac{\partial^4(I)}{\partial\zeta^4}\right]\right\}\Delta U^n$$
$$=\Delta\tau\left[\frac{\partial(E+T)^n}{\partial\xi}+\frac{\partial(F+V)^n}{\partial\eta}+\frac{\partial(G+W)^n}{\partial\zeta}\right]$$
$$-\Delta\tau\epsilon\left[h_\xi^4\frac{\partial^4U^n}{\partial\xi^4}+h_\eta^4\frac{\partial^4U^n}{\partial\eta^4}+h_\zeta^4\frac{\partial^4U^n}{\partial\zeta^4}\right]+\Delta\tau H^* \quad (4.1.20)$$

上式は BT と同様に方向毎の作用素分解が可能である. その際, BT で行った手法に加え 各作用素の主要項, 例えば第一項であれば A, を対角化しそれ以外の項をスペクトル半径によって近似する手法をとる. この対角化と近似によって各項は対角行列でもって表現されるので, U の各項に対するスカラな方程式系に帰着される. また 4 階微分の項が含まれるので, 5 重対角行列の方程式で表される.

$$\mathrm{LHS}=T_\xi^n\left[I-\Delta\tau\left\{\frac{\partial(\Lambda_\xi)^n}{\partial\xi}+\frac{\partial^2[\rho(N)^nI]}{\partial\xi^2}\epsilon h_\xi^4\frac{\partial^4(I)}{\partial\xi^4}\right\}\right](T_\xi^n)^{-1}$$
$$\times T_\eta^n\left[I-\Delta\tau\left\{\frac{\partial(\Lambda_\eta)^n}{\partial\eta}+\frac{\partial^2[\rho(Q)^nI]}{\partial\eta^2}\epsilon h_\eta^4\frac{\partial^4(I)}{\partial\eta^4}\right\}\right](T_\eta^n)^{-1}$$
$$\times T_\zeta^n\left[I-\Delta\tau\left\{\frac{\partial(\Lambda_\zeta)^n}{\partial\zeta}+\frac{\partial^2[\rho(S)^nI]}{\partial\zeta^2}\epsilon h_\zeta^4\frac{\partial^4(I)}{\partial\zeta^4}\right\}\right](T_\zeta^n)^{-1}\,\Delta U^n \quad (4.1.21)$$

ここで, $\Lambda_\xi, \Lambda_\eta, \Lambda_\zeta$ はそれぞれ行列 A, B, C を対角化したものである. 計算ステップはBTと同様に行い, 各方向の1次元内の3重対角行列の方程式求解は独立なので, 効率的な並列処理が行われる.

4.1.8 CFDアプリケーションLU

CFDアプリケーションLUは, BTでの1次近似式 (4.1.15) を次の様に

$$\mathrm{LHS} = (D^n + Y^n + Z^n)\Delta U^n \tag{4.1.22}$$

対角ブロック成分 D, 下三角ブロック成分 Y, 上三角ブロック成分 Z に分け, 次式に示すSSOR法を適用し方程式を解くプログラムである.

$$\Delta U^n = \frac{1}{\omega(2-\omega)} \left((D^n + \omega Y^n)(I + \omega (D^n)^{-1} Z^n)\right)^{-1} \mathrm{RHS} \tag{4.1.23}$$

上手法は, 次のステップによって求められる.

$$\Delta U^{n+\frac{1}{2}} = (D^n + \omega Y^n)^{-1} \mathrm{RHS} \tag{4.1.24}$$

$$\Delta U^{n+1} = \frac{1}{\omega(2-\omega)} \left(I + \omega (D^n)^{-1} Z^n\right)^{-1} \Delta U^{n+\frac{1}{2}} \tag{4.1.25}$$

LUの並列化を上に示した計算ステップの定式化のみでは困難である. なぜなら, $U_{i,j,k}$ の計算には $U_{i-1,j,k}$, $U_{i,j-1,k}$, $U_{i,j,k-1}$ が必要なので, ADEPSモデルが想定する全てのプロセッサがcoherentに動作する単純な1次元操作に収まらないからである. 何らかの制御が必要となる.

以下, LUの並列性を検証するとともにADEPSで取り扱えるような変形を行なう. 数式で取り扱うのが困難なため, プログラムによる議論を行う. 上記の計算ステップで示した通りLUの中核となる部分は 図4.7に示す3重対角ブロックの線形方程式を解くことである. 領域内のある点の値`v(i,j,k)` を求めるために必要な値とその値の依存関係は, 図4.8に示す様に強く連結している. 図4.8の灰色の点は, その隣接する黒色の点に依存していて黒色の点の値が決定するまで計算を始めることはできない.

この様な強い依存関係のもとでも, 立方体で示した領域を斜めに切断する面上の点同士の依存性はなく独立に計算することができる. 切断面は依存関係から, それに直交する方向に進行することが分かる. つまり, 図4.9に示すように, 領域を切断しながら進行する超平面上の各点を並列に計算することにより効率的な並列処理が可能となる.

今, 縦方向に延びる1次元配列にデータを分割しそれぞれをプロセッサに割り当てることを行うとする. 図4.9の下の部分に示す斜線部分は, ある時刻に計算可能なプロセッサを表している. 処理の初期段階には斜線部分は僅かであり処理効率が低いことを意味するが, 処理が進行していくと最大で $\frac{3}{4}N^2$ 個のプロセッサが計算可能な時点が存在する (今, 立方体領域の一辺を N とおいた). 平均的には N^3 の点に対して $3N$ ステップを必要とするので, プロセッサの利用率が $N^3/3N/N^2 = 1/3$ となる. ただし, プロセッサ数は一般に N^2 以下しか利用できないことが多いので, 実質的には1/3以上のプロセッサ利用率が保証されることとなる.

```
do  k = 2, nz-1
   do j = 2, ny-1
      do i = 2, nx-1
         v(i,j,k) = v(i,j,k)
                  - g * (dz(i,j,k) * v(i,j,k-1)
                         + dy(i,j,k) * v(i,j-1,k)
                         + dx(i,j,k) * v(i-1,j,k))
      enddo
   enddo
enddo
```

図 4.7: LU のコア部分

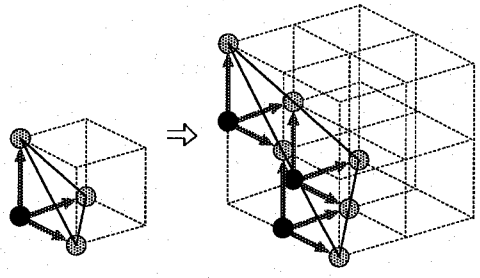


図 4.8: 格子点間の依存性, 黒で示した点は網掛け点に対して依存関係を持つ

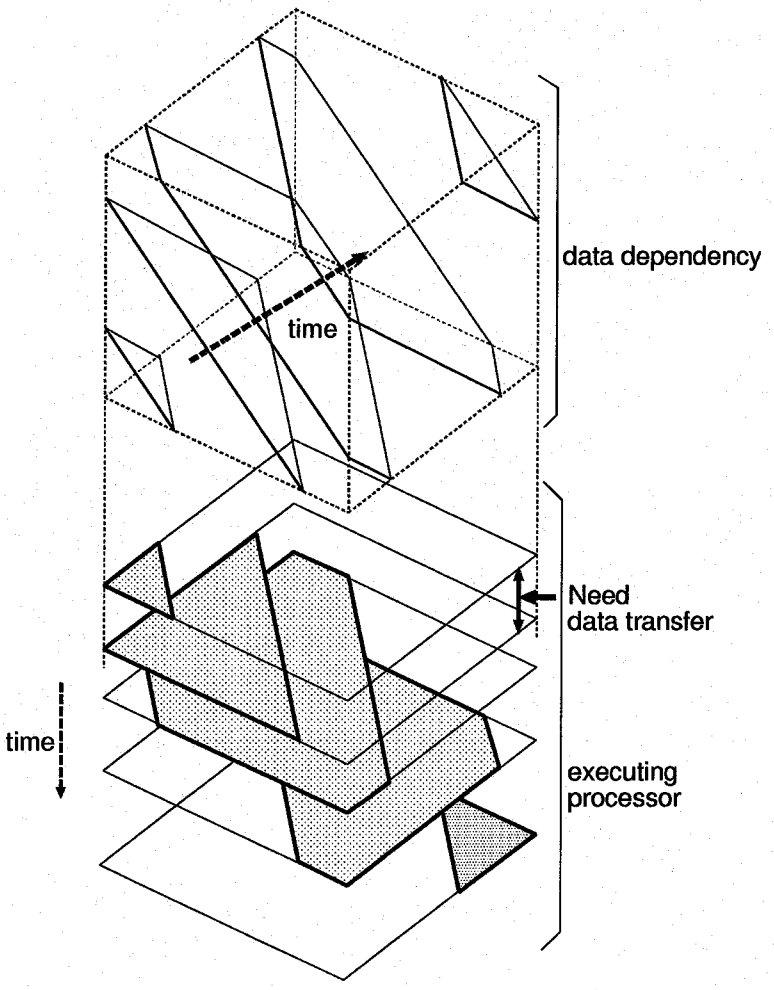


図 4.9: 超平面上での処理の進行過程, 上図の様に 3 次元データを切断した面は独立に計算可能であり, 処理の進行がその切断面 (超平面) の進行に沿って行われることを示す. 図の下部には論理プロセッサの稼働部分を示した.

4.1.9 並列計算機 ADENART 上での実験結果

以上, NPB の ADEPS モデルを元にした並列化を議論してきた. 本節では, 実際に議論した手法によりプログラムを作成し並列計算機 ADENART/256 上でそれらの経過時間を測定した結果を示す.

表 4.1は, NPB1 の 5 つのカーネルプログラムと 3 つの CFD アプリケーションを並列計算機 ADENART / 256 上で実行した結果である. なお, ADENART にはプロセッサの動作周波数が異なるモデルが存在しており, 表の上段に 34MHz, 下段に 28MHz で動作するモデルでの結果を示した. 相対的な性能比を議論するために, 標準的なスーパコンピュータとしてクレイ Y-MP の 1CPU モデルでの結果を 1 としてその比を第 2 カラムに示した.

論理的ピーク性能は ADENART/256 (34MHz) が 2.2GFLOPS であり, Y-MP/1 は約 200MFLOPS と言われている. したがって, 論理的な性能比は約 10 倍である. 表に示した結果では, 3.8 倍から殆んど同程度との結果が得られている. Y-MP はベクトル計算機であるため, メモリアクセス速度が高速であること, データ通信のためのロスがないということがその性能差の一解釈としてあげられる. また, コンパイラの最適化レベルが低いということが考えられたため, 逆アセンブルした機械語コードに対してレジスタ割り当て方法や定数式伝搬の最適化を CFD アプリケーションに対して行った. その結果を表 4.2, 表 4.3に示す. 結果から平均 10% 程度の高速化がなされることが判明し, コンパイラの最適化に高速化の余地が残っていることが分かった.

NPB では Y-MP/1 を基準とする相対性能を測るが, 並列システムに搭載されるプロセッサの性能は異なるため, あるアーキテクチャの浮動小数点計算性能を表す EP との比で正規化されるべきである. つまり, マシン X の問題 a の性能は, $(X_a/C_a)/(X_{\mathrm{EP}}/C_{\mathrm{EP}})$ で表されるべきである (ここで C は Y-MP/1 を意味する). そして, Y-MP/1 との EP 比で正規化した結果をシステム性能と呼ぶことができる. ADENART のシステム性能を, 文献 [6, 7, 36] でレポートされている様々な計算機システムとともに問題毎にプロットしたものが図 4.10である. NPB で全く同じプログラムを異なるシステムで実行した場合, システム性能はシステムの性能を反映した値を示す. また, 異なる並列化またはプログラミングモデルによって作成されたプログラムで実行した場合は, その並列化手法の優劣さも含んだ総対比を表すことになる.

分散メモリ型の並列計算機として表に挙げているものとして, CM-2, iPSC 等が比較対象となるが, ADENART が極端に悪い結果ではない. カーネルプログラムではベンダー製のライブラリの利用が許されるので, ADENART には不利な比較ではあるが, CFD アプリケーションでは比較的好成績をあげていることが分かる.

本節のまとめとして, pencil and paper の性格を持つ NPB1 の実装に対して ADETRAN を用いたプログラミングは有効な手段であり, ADETRAN が想定する並列処理モデル ADEPS が NPB 程度の広い問題範囲に対して適応可能 (表現可能) であることが示された. そして, 並列処理モデル ADEPS を効率的に実現するハードウェア ADENART 実機でのベンチーマーク実験結果からも十分にチューニングされたベンチマーク結果と対等な結果を示しており, その有効性の一端が示されたものと考えられる.

表 4.1: NAS Parallel Benchmark Results on ADENART/256

(date 1994.5.16)

	Time (sec)	Ratio to Y-MP/1 (Ratio to C90/1)	MFLOPS	Operation count	Problem size	Memory (Mw)
EP [Embarrasingly Parallel]	32.9 40.1	3.83 3.15	668 548	2.20×10^{10}	2^{28}	1
MG [Multigrid]	21.4 ---	1.03 ---	216 ---	4.63×10^{9}	256^3	130
CG [Conjugate Gradient]	10.8 16.1	1.10 .740	140 93.9	1.50×10^{9}	2.0×10^{6}	21.1
FT [3D FFT PDE]	20.3 ---	1.42 ---	277 ---	5.63×10^{9}	$256^2\times128$	113
IS [Integer Sort]	46.6 ---	.245 ---	NAN ---	NAN	2^{23}	194
LU [LU Diagonal]	327.5 399.9	1.02 (.481) .835(.394)	452 370	1.48×10^{11}	64^3	38.4
SP [Scalar Pentadiagonal]	209.9 255.4	2.25 (.880) 1.84(.723)	553 454	1.16×10^{11}	64^3	14.1
BT [Block Tridiagonal]	314.1 382.0	2.52(1.14) 2.07(.934)	719 591	2.26×10^{11}	64^3	16.1

Upper column : machine clock 34 MHz (at MATSUSHITA CO. LTD.)
Lower column : machine clock 28 MHz (at KYOTO UNIV.)

表 4.2: マシン語最適化を行ったアプリケーションプログラムのADENART/256(28MHz)での実験結果

CFD	time	MFLOPS	Ratio to Y-MP/1 (Ratio to C90/1)
LU	315	469	1.05 (.495)
SP	195	594	2.41 (.946)
BT	342	647	2.27 (1.02)

表 4.3: マシン語最適化を行ったアプリケーションプログラムのADENART/256(34MHz)を仮定した場合の評価結果

CFD	time	MFLOPS	Ratio to Y-MP/1 (Ratio to C90/1)
LU	267	554	1.25 (.589)
SP	165	703	2.83 (1.10)
BT	290	779	2.73 (1.23)

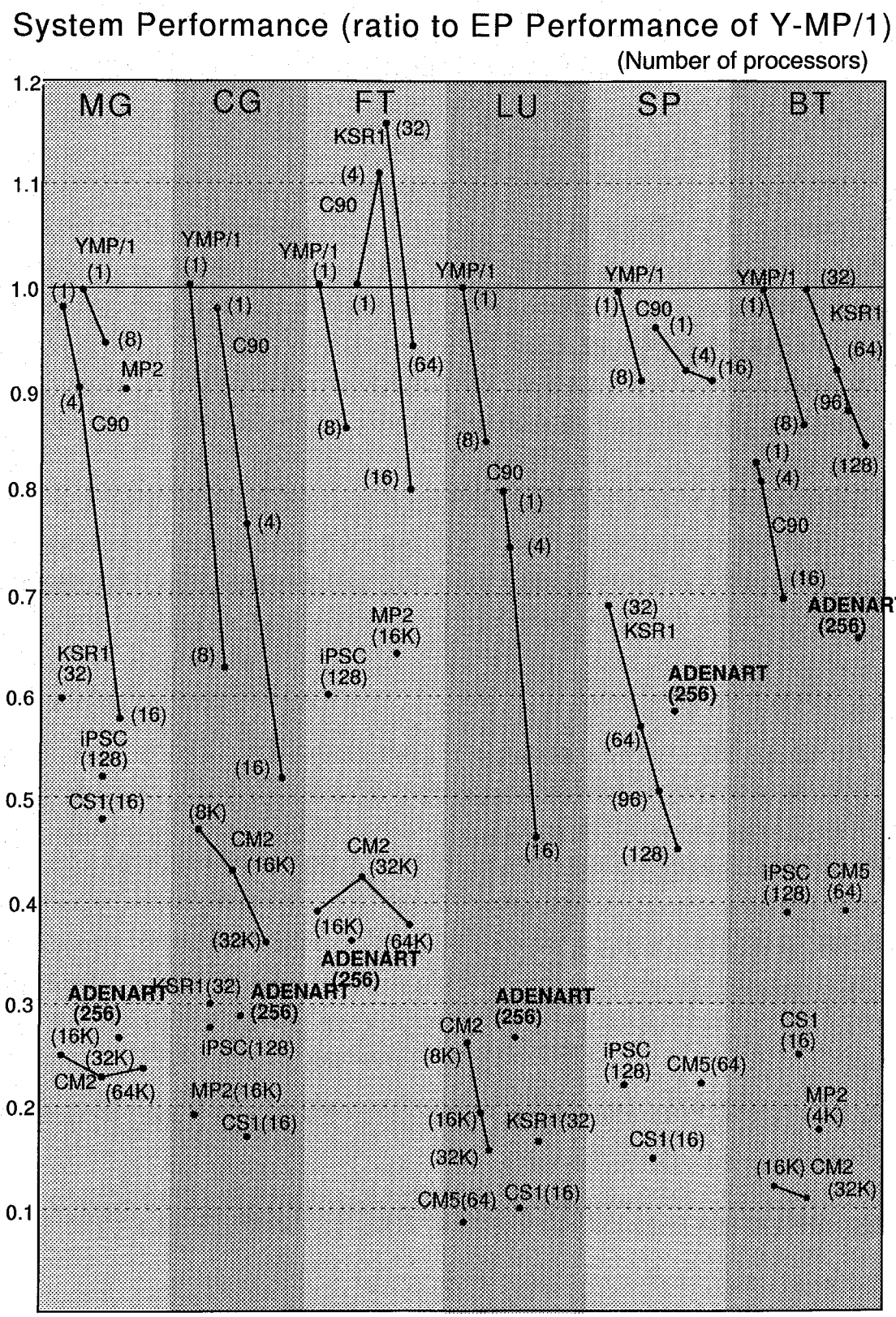


図 4.10: システム性能, マシン X の問題 a での性能を $(X_a/C_a)/(X_{EP}/C_{EP})$ で定義プロットしたもの. ここで C は CRAY Y-MP/1 を示す.

4.2 粒子プラズマコード: GYRO3D

粒子プラズマコード:GYRO3D はトカマク型核融合反応炉内のプラズマ挙動解析を行うために開発されたコードである．特に ITER などの次世代核融合実験炉の設計時に必要となるトカマク内のプラズマ全体のシミュレーションの核となる開発コードとして位置付けられている [26].
GYRO3D コードは既に日本原子力研究所, 那加研究所の徳田らによって並列化がなされ, 並列計算機 Paragon 上での長時間実験が行われている [27, 10]. 彼らの実験は, 非常に多くの計算能力を要求するため 1GFLOPS の演算の力を持つ並列計算機をもってしても数百から数千時間の計算時間が必要となる．しかしながら, 彼らが行っている実験では半径数 cm 規模のプラズマしか模擬できないため, 実際の問題を再現するには 2 倍から 10 倍規模以上のスケールにしなくてはならない．この様な計算が様々なパラメータを更新しながら実時間内に行われるためには, 数 100GFLOPS 以上の計算機が必要となる．また, 主記憶容量も数 10G バイト以上が要求されるため, 並列計算機の利用は必須のものとなっている．

本節では, GYRO3D コードを様々な並列計算機上に実装する手段としてメッセージパッシングモデルに基づく並列化と通信ライブラリ MPI を用いた実装をおこなった [21]．そして, 各種並列計算機上での実験を行いプログラムの特性, ならびに並列計算機の特性の両面について議論している．

4.2.1 GYRO3D の計算モデル

GYRO3D コードは, 一様方向に磁場をかけた 3 次元領域内の電子およびイオン (陽子など) を, 第一原理に基づいた支配方程式群を用いて時間積分して追跡するものである．このとき, 粒子の変動とともに場の量も変化するが, 両者が self-consist となる様に解いている．粒子の追跡には主に 3 段階の計算ステップを必要とする．

1. アンペールの方程式に基づく荷電粒子による電場の計算 (Charge Deposit)
2. 電位, 磁場の計算 (Field Calculation)
3. 運動方程式に基づく粒子の時間積分 (Particle Pushing)

GYRO3D では PIC(Particle In Cell) 法に基づき, 3 次元領域内に分布する粒子の相互作用を全て場の量を介して解いている．

4.2.2 領域分割と粒子分割

Charge deposit の中核部分は次式のように定式化できる

$$\rho(\mathbf{Q}) = \sum_{|\mathbf{P}-\mathbf{Q}|_{\text{lattice}}<1} w(\mathbf{P},\mathbf{Q}) \cdot q(\mathbf{P}) \qquad (\mathbf{Q} \in \text{Domain}) \tag{4.2.1}$$

ここで, ρ は場の量である電荷密度を表す. $\mathbf{Q}$ は格子点の位置, q は $\mathbf{P}$ で代表される粒子の電荷, w は粒子 $\mathbf{P}$ が点 $\mathbf{Q}$ に対して寄与する割合を定義した重み関数である. また $|\cdot|_{\text{lattice}}$ は, $|\boldsymbol{A}|_{\text{lattice}} = \max\{|A_x|, |A_y|, |A_z|\}$ でもって定義する量とする.

通常, 式 (4.2.1) をプログラムにする場合, 図 4.11にある様にループカウンタ 'I' を用いて DO-ループを構成する. ループカウンタ 'I' は粒子 $\mathbf{P}$ を識別するカウンタとしての役割をなす. 構成されたループ 4.11は不規則な間接参照から構成されていることが分かる. 特に, 粒子数がグリッドの総数を大きく越えるため, 同一アドレスへの書き込みが頻発すると容易に予想できる. したがって, charge depositing を行う DO ループは, ベクトル計算機では多数のメモリアクセス衝突を起こすために著しい性能劣化をもたらす.

次に particle pushing では, 基本方程式である運動方程式は次の様に簡略化できる.

$$m(\mathbf{P})\frac{d}{dt}\boldsymbol{v}(\mathbf{P}) = \sum_{|\mathbf{P}-\mathbf{Q}|_{\text{lattice}}<1} \boldsymbol{F}_{\text{electric}}(\mathbf{P},\mathbf{Q}) + \sum_{|\mathbf{P}-\mathbf{Q}|_{\text{lattice}}<1} \boldsymbol{F}_{\text{magnetic}}(\mathbf{P},\mathbf{Q}) \quad (4.2.2)$$

ここで m は粒子 $\mathbf{P}$ の質量, $\boldsymbol{v}$ は粒子 $\mathbf{P}$ の速度を表す. $\boldsymbol{F}$ は場から受ける外力の項である. 電荷密度項の計算式 (4.2.1) と同様に, 式 (4.2.2) の右辺もまた間接アクセスを含んでいる.

これら計算式のアクセス衝突を削減または回避するために, 幾つかの手法が提案されている [10, 21]. その手法は図 4.11と 4.12に示すプログラムの最外側ループに対してループアンローリングまたはストリップマイニングを施しているものに対応する (ここで, 図 4.11,4.12は並列化されていない逐次プログラムである).

一般的なプログラムの並列化を行う際に, データと手続きの分割は重要な要素と云われる. 領域分割 (Domain Decomposition Method, 以下 DDM) は問題領域に適した (視覚的に行うことができるという意味でも) 負荷分散またはデータ分割が行える並列処理において重要な並列化技術の一つである. また, 粒子の番号でもって分割する粒子分割法 PDM(Particle Decomposition Method) はループの単純な分割によって簡単に実現できる手法である. ただし, 全ての格子点データの複製が必要となるため PDM は領域全体がプロセッサに格納できる程度の大きさでかつ, 粒子数が非常に多い場合に適当な手法である. 一方, DDM は領域が非常に大きく粒子数が比較的少

```
DO I=1,NO
  IX=X(I); IY=Y(I); IZ=Z(I)
  IX1=IX+1;IY1=IY+1;IZ1=IZ+1
  W1=...; W2=...;   ...
  R(IX ,IY ,IZ )=R(IX ,IY ,I  )+W1*Q(I)
  R(IX1,IY ,IZ )=R(IX1,IY ,IZ )+W2*Q(I)
    ....
ENDDO
```

図 4.11: Charge Deposit の中核部分

```
DO I=1,NO
  IX=X(I); IY=Y(I); IZ=Z(I)
  IX1=IX+1;IY1=IY+1;IZ1=IZ+1
  W1=...; W2=...;   ...
  VX(I)=VX(I)+DT*(W1*RX(IX ,IY ,IZ )
       +W2*RX(IX1,IY ,IZ )+W3*RX(IX ,IY1,IZ )
    ....
  VY(I)=VY(I)+DT*(W1*RY(IX ,IY ,IZ )
       +W2*RY(IX1,IY ,IZ )+W3*RY(IX ,IY1,IZ )
    ....
    ....
ENDDO
```

図 4.12: Particle Pushing の中核部分

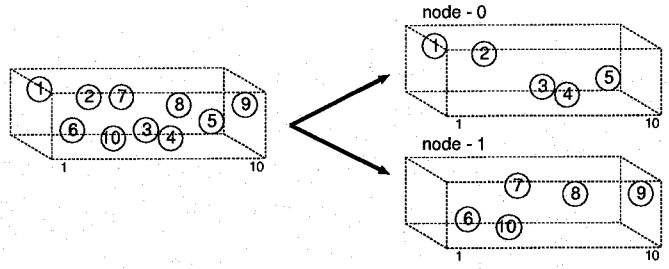


図 4.13: 粒子分割手法 (PDM)

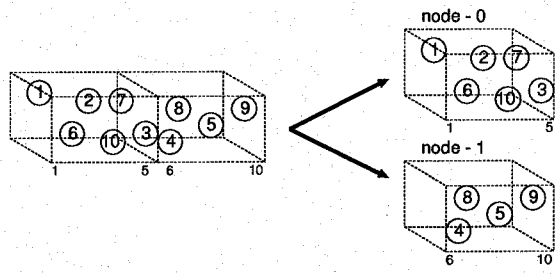


図 4.14: 領域分割手法 (DDM)

ない場合に有効な手法であるが, プロセッサを跨った粒子データの移動が頻繁に発生する場合にはデータ移動の負荷が大きくなるため得策ではない.

研究 [21] ではデータ移動を削減しつつ, DDM と PDM の効果を同時に得る手法として, 領域分割を行った後に粒子番号に基づく粒子分割を行うハイブリッドな分割手法 (D&P-DM) を提案した. この手法は次に示す DDM と PDM の両方の良い点を持っている : (i) 領域を分割するため PDM で適用可能な領域以上の大規模な問題に適用可能である, (ii) 領域の複製を作り複数のプロセッサで共有するので多くのプロセッサを使うことが可能であるということと, 粒子が移動した場合も領域を共有するためプロセッサを跨ったデータの移動量が少なくなる. したがって, プロセッサの計算能力を効果的に活かした並列処理を行うことができる.

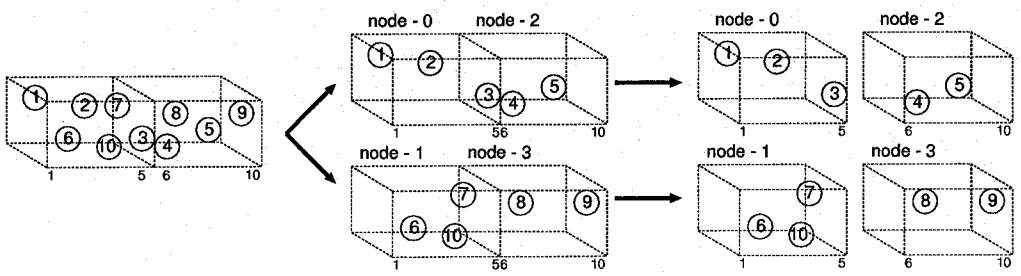


図 4.15: 領域と粒子分割の混合手法 (D&P-DM)

粒子と場の性質の異なる量をADEPSで扱う場合, 同一のグリッド分割は適用できないため一方 (粒子) をベクトルとして扱いADEPSの範疇で表現することは可能である. ただし, 領域を越える粒子の運動は粒子属性データの移動を意味しており, 何らかの方法でのデータ転送が必要となる. 個々のプロセッサで移動する粒子数が同一個数であれば, pcast 文等でデータの移動を記述可能であるが, 不規則であり一対一の通信形態が必要となるのでメッセージパッシングの枠を取り込まなくてはならない. いずれにしても, MPI+Fortran90の実装系では可能な範囲であることには間違いない.

4.2.3 並列高速フーリエ変換

FFTの並列化には幾つかの手法が存在する. 3次元データのフーリエ変換はADEPSならびにNPBの部分で, 解説した通り x-,x-,z- 方向への交互方向の分解が有効である. 例えばx方向の処理を行う場合には, 3次元配列を第一インデックスで表現される一次元配列の集合と見なし, 各1次元配列のフーリエ変換を並列に行う. 図4.16に, 交互方向に切り替えて行う3次元データのフーリエ変換の例を示す. X軸方向の処理を行った後には, 処理する方向を切り替えて別の方向の処理を行う. ここで示した方向の切り替えは転置もしくは回転に対応している. 分散メモリシステム上で転置, 回転を行うためには全対全のストライド付転送が必要となるため, 並列フーリエ変換の大きい部分を占めることとなる. ここに示した, 交互方向のFFT操作はADETRAN4を用いることで次のように極めて簡単なプログラム表記で実現可能である.

```
pdo  j=1,n2, k=1,n3
   call fft1d(a(1,/j,k/),b(1,/j,k/),n1)
pend
pdo  k=1,n3, i=1,n1
   call fft1d(b(i/,1,/k),c(i/,1,/k),n2)
pend
pdo  i=1,n1, j=1,n2
   call fft1d(c(/i,j/,1),d(/i,j/,1),n3)
pend
```

FFTの並列化の別手法として, 同時に変換可能な物理量を並列に処理する方法が提案できる. GYRO3Dの物理モデルの場合, 2つの物理量と6つの物理量を同時にフーリエ変換することができる (また同時に逆フーリエ変換も). 本手法を適応する場合は空間全対のデータを分割すること無く一つのプロセッサに保持させなくてはならないので, DDMを採用した場合には不適当で, PDMの場合に有効な手法と言える.

同時変換を行う手法は, 並列処理する際に割り当てられるプロセッサの数が同時変換可能な物理量の数に限られるという問題が存在する. この問題を解決する手法として, PDMとDDMを組み合わせたと同様の手法が提案できる. 同時変換可能な物理量のフーリエ変換に複数のプロセッサを割り当てることによって並列効果を向上させることが可能となる. ここで, MPIを使った実装を考慮した場合には, 容易にこの手法が実装できることを追記しておく. MPIにはMPIプロセス

を柔軟にグループ化できる機能を有しており, $(N_x \times N_y)$ に配置した論理的プロセッサグループを $(\frac{N_x}{2} \times N_y) \times 2$ の様に 2 つのプロセッサグループに分割することができる. さらに, 分割されたプロセッサグループの中で新たなプロセッサ識別子が定義され, サブプロセッサグループ内の MPI プロセス割り当てられるので, プロセッサグループの中で閉じて, またプロセッサ数可変な 3 次元フーリエ変換を行うサブプログラムを開発しておけば容易に実装が可能である (図 4.17).

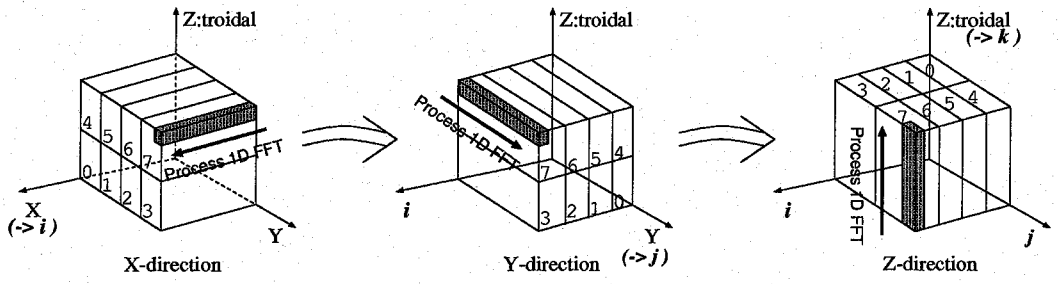


図 4.16: 交互方向操作に基づく並列 FFT

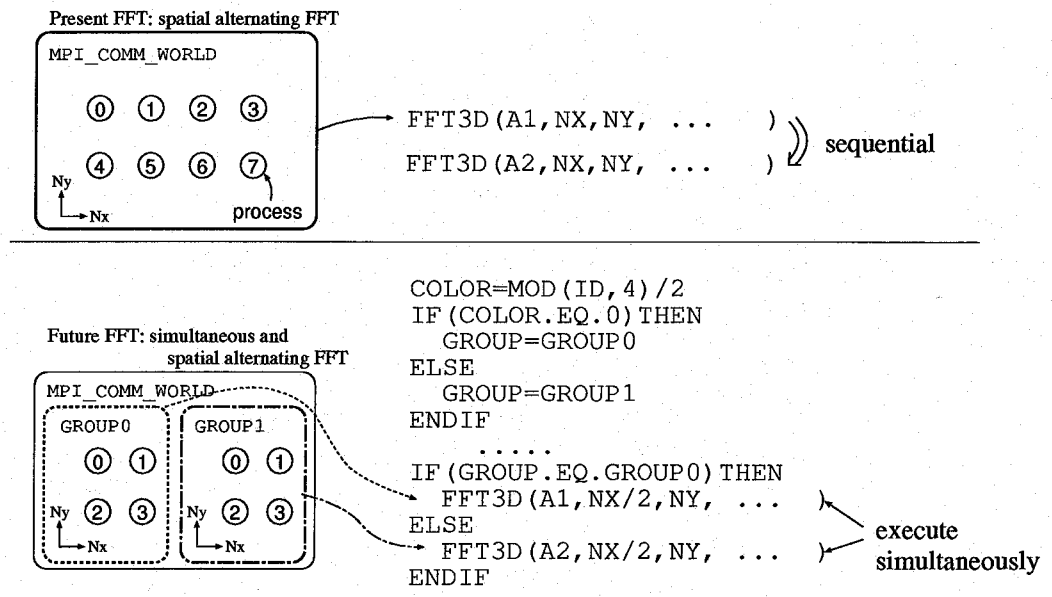


図 4.17: 交互方向操作 (上) と同時操作の混合型 FFT(下), 8 プロセッサを (4×2) に分割配置し領域にマッピングし単純に 1FFT 操作を順に行うものと 4 プロセッサごとに (2×2) に分割配置しそれぞれ別々の配列にマッピングし FFT 操作を 2 つ同時に行う例を示している.

4.2.4 周波数分割

3次元ポアソン方程式: $\Delta\phi = \rho$ はフーリエ変換を行うことによって, 次の様な周波数領域の問題に帰着される

$$-|i|^2\tilde{\phi}_i = \tilde{\rho}_i, \qquad i = (i,j,k) \tag{4.2.3}$$

上式から, 各周波数 i の計算は独立であり並列処理可能である. 並列にフーリエ変換を行った後に, 結果として分散して保持されている周波数を保持しているプロセッサ上で計算し, 直ちに逆変換に移れるためデータの分割方法に特に依存せず効率的な並列処理が可能である.

4.2.5 並列計算機上の数値実験結果

GYRO3D コードを Fortran90 と MPI を用いて, 先に示した並列化手法を用いて並列化コードを開発した. 本節では日立 SR2201, 富士通 VPP300, 日本電気 SX4, IBM SP2, インテルパラゴン上で実行した結果を示す.

数値実験に使用したパラメータは次のものである.

1. グリッドサイズは $32 \times 32 \times 16$.
2. 粒子数は電子及びイオン併せて 1,048,576($= 2^{20}$) 個
   セルの中には平均 64 粒子が存在.
3. 100 時間ステップの時間積分を行った.

図 4.18に, CPU 時間ではなくプログラム実行に要した経過時間を示した. 図 4.18の縦軸は経過時間 (秒) を表し, 横軸は使用したプロセッサ数を意味する. 経過時間には データの初期化と計算結果の出力が含まれるが, プログラムの読み込み時間やプロセスの fork,join の時間は含まれない. また, 領域分割を効率的に行うために, 使用したプロセッサ数は 2 のベキ (1, 2, 4, 8, 16, 32, ...) としたが, 計算機に搭載される主記憶量の制限により計算機によっては少数プロセッサ使用時に計測できないものが存在した. 各実測点の下には経過時間の数値も付記している. さらに, VPP300, SX4 等のベクトル計算機では, アクセス競合を回避するようにプロセッサ内のベクトル最適化を実施している. 他のスカラ計算機では領域分割のみを使用した. SR2201 においては, ループ内部に含まれる実行文が多いために普通では疑似ベクトル化が行われなかった. 実行文を分けてループを再構成し疑似ベクトル化したものも追記している.

図 4.18から, 並列化した GYRO3D コードは機種を問わず幅広い範囲でプロセッサ数に対する良好な速度向上比と並列化効率を示している. 計測結果は 3 つの帯をなしており, 丁度使用した計算機のプロセッサの世代に対応している. 一番上に位置するパラゴンは 1 プロセッサ当り 75M-FLOPS の i860 を使用している. 中段に位置する SR2201,SP2 はそれぞれ 1 プロセッサ当り 300 MFLOPS の PA-RISC(ただし疑似ベクトル:PVPSW[28] の改良が加えられている) と 1 プロセッ

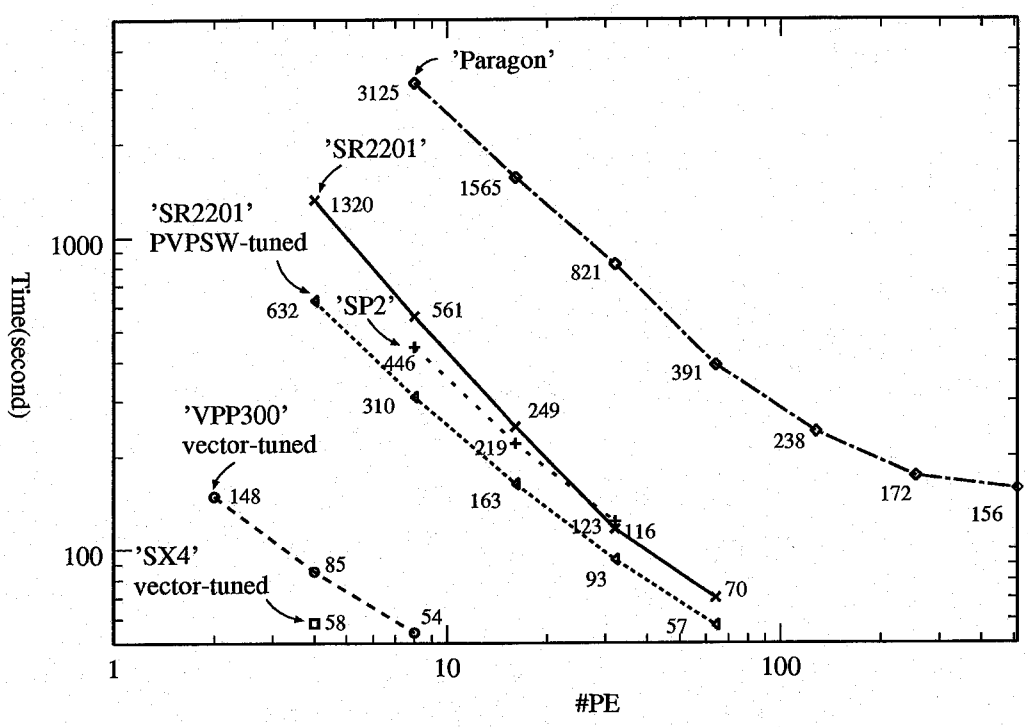


図 4.18: 種々の並列計算機における GYRO3D の実験結果, 使用したプロセッサ数 (横軸) に対する経過時間 (縦軸, 単位は秒) をプロットしている. 両軸に対数スケールを用いている.

表 4.4: 主要 3 サブルーチン (C.D., F.C., P.P.) のコスト分布

表 3.1: SR2201

#PE	4	8	16	32	64
C.D.	536	226	85	34	20
F.C.	30	20	12	13	17
P.P.	740	314	149	64	26

表 3.2: SR2201(PVPSW tuned)

#PE	4	8	16	32	64
C.D.	251	119	54	28	14
F.C.	27	18	14	11	15
P.P.	341	158	90	47	21

表 3.3: SP2

#PE	8	16	32
C.D.	175	83	46
F.C.	19	14	12
P.P.	248	118	61

表 3.4: VPP300(Vector tuned)

#PE	2	4	8
C.D.	44	24	13
F.C.	8	7	7
P.P.	64	33	19

表 3.5: Paragon

#PE	8	16	32	64	128	256	512
C.D.	825	406	206	86	50	26	20
F.C.	61	38	24	19	21	27	34
P.P.	2126	1031	509	231	110	50	33

C.D. ··· Charge Deposit
F.C. ··· Field Calculation
P.P. ··· Particle Pushing

サ当り 266 MFLOPS の POWER2 を使用している．下段はベクトルプロセッサであり，VPP300 は 1 プロセッサ当り 2.2GFLOPS, SX4 は 1 プロセッサ当り 2.0GFLOPS を使用している．プロセッサの開発された年代では，上段が 1980 代後半，中段が 1990 代中盤の RISC チップ，下段が 1990 代中盤の CMOS ベクトルチップということになる．

次に各機種ごとの計測結果を見ていくと，パラゴンでは 128 プロセッサまでの速度向上はほぼ線形に向上しているが，256 プロセッサ以降は並列化効率は著しく低下していることが分かる．また，プロセッサ数が 32 から 64 になるときに速度向上比が線形を越えたスーパリニア状態になっている．

同様に，疑似ベクトル化によるチューニングを行わない SR2201 の結果にも線形を越えたスーパリニア状態が観測することができる．疑似ベクトルを行った場合には，ほぼ線形の速度向上を示すことが確認できる．疑似ベクトルを施すことによる性能向上率は 1.5 から 2 倍程度である．

SP2 は比較的素直な速度向上を示している．

ベクトル計算機である VPP300 では，やや速度向上は鈍いものの十分許容できるものが得られていると言えよう．

SX4 は，1 プロセッサ当りのピーク性能は VPP300 には 1 割程度劣るもののプログラムの実行性能では大きく上回る結果を得ている．

さらに，詳しい性能分析を行うために GYRO3D の中の 3 つの主要サブルーチンに要したコスト分布を調べたものを表 4.4(1-5) に示した．表から，GYRO3D の最も計算量が多い部分は Particle Pushing(P.P.) の部分であり，次に Charge Deposit(C.D.) が負荷が大きく，Field Calculation(F.C.) は殆んど無視できる部分であることが分かる．計算量が大きい P.P. や D.C. は機種に依存すること無く，プロセッサ数に応じて高速化されている．パラゴンの 512 プロセッサの場合を除いて殆んどプロセッサ数に比例した速度向上を見せている．しかしながら，F.C. はプロセッサ数に比例するどころか殆んど効果のないものが多い．

表 4.4に示した主要 3 サブルーチン毎の実行時間を総和したものと図 4.18の結果とが著しく異なるものが存在しているが，主要 3 ルーチン以外の初期化，計算結果出力の部分が大きなコストになっているからである．実際，図 4.18の値との差は初期化部分が 90% 以上を占めている．初期化の内部では，乱数を発生させ粒子の初期状態を決めるわけであるが，乱数発生方法において並列化できない手法がとられているため，全てのプロセッサで冗長に計算を行い，担当するプロセッサにのみ値を反映させる方式をとっている．そのために，プロセッサ数が増加しても，逐次処理を行っているのと等価となりプログラム全体の並列化効率を下げるという結果になっている．

初期化の問題は，今回行ったテストプログラムに限って起こる問題であるが，実際のプロダクションランを行う場合にはもっと深刻な問題が発生する．通常プロダクションランを行う場合には，一回に利用可能なシステムの使用時間が限られておりタイムアウト前にメモリ上の全データを主記憶から磁気記憶装置に保存し次回のリスタートに備えることを行う．GYRO3D の場合リスタートに必要なデータは，今回測定した問題規模ではシステム全体で 300M バイトを越える．徳田らが行っている実験では，その 10 倍の 3G バイトを越える場合も実際にある．一般に並列計算機

システムは全てのプロセッサが磁気記憶装置を接続していないため, それを接続しているプロセッサにOSがデータを転送し記録を行わなくてはならない. つまり, 並列に記録できないシステムが殆んどである. プロダクションランでは十数時間プログラムを実行できてもそのうち1時間は, リスタートのための入出力に要することは珍しくはない.

研究 [21] では並列入出力に関する議論は行わなかったが, 今後大規模アプリケーションの並列処理を行う上で重要な課題であることが浮彫りとなったといえる.

本節のまとめとして, 実アプリケーションの並列処理の一例として粒子プラズマコードGYRO3Dの並列化ならびに各種計算機での実測を行った. 並列化において, 並列処理モデルADPESに基づく交互方向のFFTの効果的な記述が可能であることが示された. また, 領域分割ならびに粒子分割の併用による高並列処理の良好な並列化効率を得ることができた.

第 5 章

結言

本研究では, 分散メモリ型並列計算機の処理系とアルゴリズムの側面から研究を行ってきた. これら研究結果と得られた知見は以下の様にまとめられる.

第 2 章では, 並列アルゴリズムの表現ならびに計算機実験において利用する並列計算言語 ADETRAN4 の実装を行った. 実装した並列計算機は VPP-500 および MPI と Fortran90 が利用できる計算機である. また, 分散メモリ型並列計算機上での並列アルゴリズムを柔軟に記述可能にするための構文拡張と他の言語とのインターフェイスを明確化した.

- 並列言語 ADETRAN4 の処理系を MPI と Fortran90 が利用できる計算機上で利用可能としたことで, 事実上殆んど全ての計算機上で ADETRAN4 プログラム共有が可能となった. また, 他言語とのインタフェースが明確化されたことで, すでに他の言語で開発された並列プログラムの関数呼び出しや, その逆も可能となった.

第 3 章では, 行列計算の典型的な例としてあげられる線形方程式求解のための LU 分解, 固有値計算で利用されるハウスホルダー 3 重対角化, 固有値探索のための divide and conquer 法を取り上げ, 並列化並びにコストモデルの導出を通じてその並列処理の効果を調べるとともに新たなアルゴリズムの検証を行った.

- LU 分解の並列化では縦ブロックに基づく分割, つまりブロックサイクリック分割による LU 分解の実装とそのコストモデルを算出し, 並列計算機の性能と問題サイズから並列計算機システムの資源を有効に使う分割方法を導出することができた.

- また, ハウスホルダー 3 重対角化では行列の対称性を活かしたアルゴリズムを提案し, 従来の手法とコストモデルでの比較を行い優れた手法であることを示した. さらに, 従来並列計算機で効果があるとされてきたブロック化手法について, それが効果的であるための条件を示した.

- divide and conquer 法の並列化を行い, 従来法と比較して 1.3 倍高速であることを実測するとともに, さらに改良を行ったアルゴリズムで 1.4 倍高速であることを示した.

第4章では, NASパラレルベンチマークと粒子プラズマコードの2つを取り上げ 大規模問題に対する並列化の手法を提案した. 特に粒子プラズマコードでは, 粒子コード特有の計算部分の並列化の困難さを示すとともに粒子分割, 領域分割に基づく並列化手法を示し, それを用い並列化したプログラムの数値実験を行った.

- 並列処理モデル ADEPS に基づくNASパラレルベンチマークの並列化を行い, 並列計算機 ADENART 上で測定しシステム性能比に基づく比較を行ったところ, 同時期に出現した分散メモリ型で十分にチューニングされた結果とほぼ同程度の結果を得た. このことから, ADEPSに基づく並列化およびADENARTのアーキテクチャの優位性が示された.
- 粒子プラズマコードでは, ADEPSに基づく高速フーリエ変換と粒子運動の並列化を行ったが, 問題に依存する別の観点からの並列性と組み合わせた並列手法が有効である結論を得た. 複数の交互方向FFT操作をグループに別けて同時に行うものと, 領域分割と粒子分割が有効であった.

以上, 本研究は並列計算言語 ADETRAN4 を中心に, その処理系を利用した数値計算における並列アルゴリズムの提案, 評価を行ってきた. 本研究で開発した処理系のように明確な並列モデルを持った処理系が, 将来自動並列化コンパイラにおいて中間言語的な役割を果たすと考える. また, コストモデルによる評価手法は, 並列計算機上のコンパイラ内部の解析や数値ライブラリでの適用が大いに期待できる. 今後, コンパイラや数値ライブラリが容易にかつ自動的にコスト見積もりを行う様な技術確立が必要であろう.

謝 辞

本論文をまとめるにあたり，終始御懇切な御指導と御鞭撻を賜りました京都大学工学研究科教授島崎眞昭博士に謝意を表します．

また本論文について数々の有益な御教示ならびに御助言を頂きました京都大学情報学研究科教授富田眞治博士に心より感謝致します．

また日頃，本研究の御指導と本論文作成に対する御助言を頂きました京都大学情報学研究科教授野木達夫博士に深く感謝の意を表します．

本研究の遂行にあたり御指導頂きました日本原子力研究所浅井清理事，同研究所安全性試験研究センター 秋元正幸センター長，同研究所計算科学技術推進センター次長 相川裕史博士，同研究所同センター並列処理基本システム開発グループリーダ 平山俊雄博士に深く感謝致します．

本研究を行うにあたり御尽力頂いた京都大学情報学研究科助手 原田健自博士，日本原子力研究所計算科学技術推進センター並列処理基本システム開発グループ樋口健二副主任研究員，武宮博氏，小出洋博士に厚く御礼申し上げます．

研究業績一覧

発表論文

1). 今村俊幸: 分散メモリ型並列計算機における縦ブロック分割並列 LU 分解, 日本応用数理学会論文誌, Vol.8, No.3 (1998)

2). T. Imamura: An Estimation of Complexity and Computational Costs for Vertical Block-Cyclic Distributed Parallel LU Factorization, The Journal of Supercomputing, Vol.15, No.1, (2000)

3). 今村俊幸 : 並列言語 ADETRAN4 での行列計算 —VPP500 上の処理系実装とその応用 —, 並列処理シンポジウム JSPP'96 (1996)

4). T. Imamura, T. Nogi, K. Sakai and Y. Obayashi: NAS Parallel Benchmark Results on ADENART, Parallel Computational Fluid Dynamics: New Algorithms and Applications, pp.161 (1995)

5). T.Imamura, S.Tokuda and H.Naitou: Parallelization of 3D Gyrokinetic Particle Code: GYRO3D using MPI and its Performance evaluation on Parallel Computers, Proceedings of Joint Internatinal Conference on Mathematical Methods and Supercomputing for Nuclear Applications, Vol.2 (1997)

6). 武宮博, 樋口健二, 本間一朗, 太田浩史, 川崎琢治, 今村俊幸, 小出洋, 秋元正幸: 並列プログラミング支援環境の現状と動向, JAERI-Review, 97-005 (1997)

7). 今村俊幸, 小出洋, 武宮博: 異機種並列計算機間ライブラリ: Stampi — 利用手引書, JAERI-Data/Code, 98-034 (1998)

国内学会発表, 研究会

1). 今村俊幸 : 並列計算機 ADENART による実対称行列の固有値計算について, 日本応用数理学会平成 4 年度年会 (1992)

2). 今村俊幸 : 超並列計算機による高速固有値計算法, 平成 4 年度名古屋大学太陽地球環境研究所「総合解析」合同シンポジウム (1993)

3). 今村俊幸 杉山和徳, 野木達夫 : 並列言語 ADETRAN4 の VPP-500 での利用と評価, 情報処理学会 HPC 研究会 (1995)

4). 杉山和徳, 今村俊幸 野木達夫 : 仕様記述言語 VDM による並列計算機 ADENA4 シミュレータの実現情報処理学会 HPC 研究会 (1995)

5). 今村俊幸 : 並列言語 ADETRAN4 の VPP-500 での利用と評価, 京都大学計算機センター第 49 回研究セミナー報告 (1996)

6). 今村俊幸 : 複合並列計算機における ADETRAN4 処理系の実装について, 情報処理学会 HPC 研究会 SWoPP96(1996)

7). 徳田伸二, 小出洋, 今村俊幸, 松本太郎, 小境博 : トカマク・プラズマにおけるハイブリッド・シミュレーション, 日本物理学会第 53 回年会, 1pP5(1998)

8). 今村俊幸 : ベクトル並列計算機での固有値ライブラリ, 日本応用数理学会平成 11 年度年会, (1999)

参考文献

[1] J. C. Adams, et al., Fortran 95 Handbook, Complete ISO/ANSI Reference, McGrawHill, 1997.

[2] A. Anderson, et al., LAPACK: Portable Linear Algebla Library for High-Performance Computers, LAPACK Working Note 20, University of Tennessee, CS–90–105, 1990.

[3] P. Arbenz, W. Gander and G. H. Golub, Restricted rank modification of the symmetric eigenvalue problem: theoretical considerations, Linear Algibra and its Application. 104, pp.75–95, 1988.

[4] D. Bailey, et al., The NAS Parallel Benchmarks, Tech. Report RNR-91-02, NASA Ames Research Center, 1991.

[5] D. Bailey, et al., The NAS Parallel Benchmarks 2.0, Tech. Report NAS-95-020, NASA Ames Research Center, 1995.

[6] D. Bailey, et al., NAS Parallel Benchmark Results, IEEE Parallel & Distributed Technology, Vol.1, No.1, pp.43-51, 1993.

[7] D. Bailey, et al., NAS Parallel Benchmark Results Update, Tech. Report RNR-94-006, NASA Ames Research Center, 1994.

[8] L. S. Blackford, et al., ScaLAPACK: A Linear Algebra Library for Message-Passing, SIAM Conference on Parallel Processing, 1997.

[9] J. J. M. Cuppen, A divide and conquer method for the symmetric tridiagonal eigenproblem, Numer. Math. 36, pp.177–195, 1981.

[10] V. K. Decyk, H. Naitou, T. Sonoda and S. Tokuda, Particle-in-Cell Simulation on the Fujitsu VPP500 Parallel Computer, Supercomputer(to appear).

[11] F. Desprez, B. Tourancheau and J. J. Dongarra, Performance Complexity of LU Factorization with Efficient Pipeling and Overlap on a Multiprocessor, Tech. Rep. Univ. Tennessee, CS-93-218, 1994.

[12] J. J. Dongarra, Performance of Various Computers Using Standard Linear Equations Software, Tech. Rep. Univ. Tennessee, CS-89-85, 1997.

[13] J. J. Dongarra, H. W. Meuer and E. Strohmainer, TOP500 Report 1996, Supercomputer, Vol.13, No.1, 1997.

[14] J. J. Dongarra and D. C. Sorrensen, A fully parallel algorithm for the symmetric eigenvalue problem, SIAM J. Sci. Stat. Comput., Vol.4, No.2, pp.139–154, 1987.

[15] J. J. Dongarra and R. A. van de Geijn, Reduction to condesed form to the eigenvalue problem on distributed memory architectures Parallel Computing, Vol.18, No.9, pp.973–982, 1992.

[16] G. H. Golub and C. F. Van Loan, Matrix Computations, 3rd edition, The Johns Hopkins University Press, 1996.

[17] High Performance Fortran Forum, High Performance Fortran Language Specification, Version 1.0, 1993.

[18] High Performance Fortran Forum, High Performance Fortran Language Specification version 2.0, 1997.

[19] T. Imamura, An Estimation of Complexity and Computational Costs for Vertical Block-Cyclic Distributed Parallel LU Factorization, The Journal of Supercomputing, accepted.

[20] T. Imamura, etc at., NAS Parallel Benchmark Results on ADENART, Parallel Computational Fluid Dynamics(Proc. Intl. Conf. PCFD'94), pp.161–168, Elsevier, 1995.

[21] T. Imamura, S. Tokuda and H. Naito, Parallelization of 3D Gyrokinetic Paricle Code: GYRO3D using MPI and its Performance Evaluation on Parallel Computers, Proc. Joint Intl. Conf. M&C+SNA'97, Vol.2, 1997.

[22] H. Kadota, et al., VLSI Parallel Computer with Data Transfer Network: ADENA, Proc. 1989 Int. Conf. Parallel Processing, I, pp.319-322, 1989.

[23] K. Kennedy, et al., Fortran D Language Sepcification, CRPC-TR 90079, Rice University. 1990.

[24] Message Passing Interface Forum, MPI: A Message Passing Interface Standard, 1995.

[25] Message Passing Interface Forum, MPI-2: Extensions to the Message-Passing Interface, 1997.

[26] H. Naitou, NEXT project and Gyrokinetic Particle Simulation, J. Plasma and Fusion Research, No. 70, pp.135, 1994.

[27] H. Naitou, T. Sonoda, S. Tokuda and V. K. Decyk, Parallelization of Gyrokinetic Particle Code and its Application to Internal Kink Mode Simulation, J. Plasma and Fusion Research, No. 72, pp.3, 1995.

[28] H. Nakamura, et al., A Scalar Architecture for Pseudo Vector Processing based on Slide-Windowed Registers, Proc. ACM Supercomputing '93, 1993.

[29] T. Nogi, Parallel machine ADINA, Computing Methods in Applied Science and Engineering, pp.103–122, North-Holland, 1982.

[30] T. Nogi, Prallel programming language ADETRAN, Memories of the Faculty of Engineering, Kyoto University, Vol.51, No.4, pp.235-289, 1989.

[31] T. Nogi, Parallel Computation on ADENA, Parallel Computing '91, pp.619-626, 1992.

[32] T. Nogi, ADENA Computer IV, Memories of the Faculty of Engineering, Kyoto University, Vol.55, No.1, pp.21-36, 1993.

[33] T. Nogi, Promissing Data Parallel Environment —ADEPS, ADETRAN, ADENA—, Proc. 1st Aizu Int. Sympo. Parallel Algorithms/Architecture Synthesis, pAs'95, pp.45–53, IEEE Computer Society, 1995.

[34] S. Odanaka, et al., Massively Parallel Computation Using a Splitting-up Operator Method for a Three-dimensional Device Simulation, IEICE Technical Report, 10, 1992.

[35] OpenMP Architecture Review Board, OpenMP Specifications, `http://www.openmp.org/`, 1997.

[36] S. Saini and D. H. Bailey, NAS Parallel Benchmark Results 12-95, Technical Report NAS-95-021, NASA Ames Research Center, 1995.

[37] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema and C. R. Moler, Matrix Eigensystem Routines – EISPACK Guide, Lecture Note in Computer Science 6. 2nd ed., Springer Verlag, 1976.

[38] C. W. Tseng, An Optimizing FORTRAN D Compiler for MIMD Distributed Memory Machine, Ph.D thesis, Rice Univ., 1993.

[39] D. C. Watkins, Fundamentals of Matrix Computations, John Wiley & Sons,Inc, 1991.

[40] H. F. Weinberger, Variational methods for eigenvalue approximation, Regional Conference Series in Applied Mathematics 15, SIAM, Philadelphia, 1974.

[41] K. Zaiki, et al., Parallel Programming Language ADETRAN, Parallel Language and Compiler Research in Japan, Kluwer Academic, 1995.

[42] H. Zima, et al., Vienna Fortran —A Language Specification Version1.1, ACPC/TR 92-4, 1992.

[43] 石川 達也, ADETRAN コンバータ ADECONV とその Sorting 問題への応用, 修士論文, 京都大学大学院工学研究科応用システム科学専攻, 1996.

[44] 今村 俊幸, 並列言語 ADETRAN4 の VPP-500 での利用と評価, 京都大学計算機センター, 第 49 回研究セミナー報告, 1996.

[45] 今村 俊幸, 並列言語 ADETRNA4 での行列計算 —VPP500 上の処理系実装とその応用 —, 並列処理シンポジウム JSPP'96, pp.17–24, 1996.

[46] 今村 俊幸, 複合並列計算機における ADETRAN4 処理系の実装について, 情報処理学会研究報告, Vol.96, No. 81, pp.13–18, 1996.

[47] 今村 俊幸, 分散メモリ型並列計算機における縦ブロック分割並列 LU 分解, 日本応用数理学会論文誌, Vol.8, No. 3, pp.57–72, 1998.

[48] 今村俊幸, 杉山和徳, 野木達夫, 並列言語 ADETRAN 4 の VPP-500 での利用と評価, 情報処理学会研究報告, 95-HPC-58, pp.13–18, 1995.

[49] 岩下, 進藤, 岡田 : VPP Fortran : 分散メモリ型並列計算機言語, 並列処理シンポジウム JSPP'94, 1994.

[50] 岡田, 坂本, 浅井 : VPP500 システムのソフトウェア, 電気情報通信学会論文誌, D-I, Vol. J78-D-I, No.2, pp.149–161, 1995.

[51] 岡本 理 他, 並列計算機 ADENART のシミュレーション分野での応用, National Technical Report, Vol.39, No.1, 1993.

[52] 小国 力 編著, 村田 健朗, 三好 俊郎, J.J.Dongarra, 長谷川 秀彦, 行列計算ソフトウェア. WS, スーパーコン, 並列計算機, 丸善, 1991.

[53] 黒田 茂 他, 並列計算機 ADENA 用自動並列化コンパイラ, 第 42 回情報処理全国大会, vol.5, pp.119–120, 1991.

[54] 佐々 政孝, プログラミング言語処理系, 岩波講座ソフトウェア科学 5, 岩波書店, 1989.

[55] 寒川 光, RISC 超高速化プログラミング技法, 共立出版, 1995.

[56] 建部 修見, 分散メモリ型並列計算機による LU 分解, 情報処理学会研究報告, Vol. 95, pp.55–66, 1995.

[57] 戸川 隼人, マトリクスの数値計算, オーム社書店, 1971.

[58] 富田 眞治, 末吉 敏則, 並列処理マシン, オーム社, 1989.

[59] 中谷, HPF コンパイラの現状と課題, 電気情報通信学会論文誌, D-I, Vol. J78-D-I, No.2, pp.142–148. 1995.

[60] 野木 達夫, 並列計算モデル PB 対 ADEPS, 応用数理, Vol.3, No.3, 1993.

[61] 野木 達夫, 偏微分方程式の数値解析と並列計算, 地球環境シミュレーターに向けて, 数理科学, No. 417, 1998.

[62] 別府 良孝, スーパーコンピュータに適した固有値ルーチン, bit 増刊号「スーパーコンピュータと大型数値計算」, pp.96–105, 共立出版, 1987.

[63] 若谷 彰良, 野木 達夫, 並列計算機用言語 ADETRAN 処理系の開発, 情報処理学会全国大会, pp.153–154, 1988.

付録 A

ADETRAN4 文法規則

本章では本論文での議論を補うため, 並列言語 ADETRAN4 の文法を規定する BNF(Backus-Naur Form) を示す. ここで BNF 記法として次のルールを採用している.

1. < > 文法記号を定義する. この定義によりトークンと文法記号を区別する.
2. ::= 文法の書き換え規則を意味する.
3. | 文法の書き換えの別の方法を意味する.
4. [ ] オプションを意味する (紙面の都合上利用した).
5. ‘ ’ トークンの定義.
6. " " 正規表現によるトークン定義.
7. /* ... */ コメント

] ... を使ってオプションの反復を記述せずに, 回帰的な生成規則または正規表現を使って記述するようにした. トークンの大文字小文字の区別は無いので全て大文字で統一した.

```
/*
        Simplified BNF of ADETRAN4
                                BY Toshiyuki Imamura.

        NOTE:: double-quoted string = regular expression
               single-quoted string = token
                                                                    */


/* ADETRAN4 term and concepts */
<ADETRAN4_program> ::=
                    <g_l_subprogram>
                  | <ADETRAN4_program> <g_l_subprogram>

<g_l_subprogram> ::=
                    <g_subprogram>
```

```
                | <l_subprogram>

<g_subprogram> ::=
                  <g_subroutine>

<l_subprogram> ::=
                  <l_subroutine>
                | <l_function>


/* procedure */
<g_subroutine> ::=
                  'GSUBROUTINE' <proc_id> <g_arg> <g_body> 'END'

<l_subroutine> ::=
                   'LSUBROUTINE' <proc_id> <l_arg> <l_body> 'END'

<l_function> ::=
                  [ <type> ] 'LFUNCTION' <proc_id> <l_arg> <l_body> 'END'

<g_arg> ::=
                <g_l_arg>

<l_arg> ::=
                <g_l_arg>

<g_l_arg> ::=
                  [ '(' [ <g_l_arg_list> ] ')' ]

<g_l_arg_list> ::=
                  <id>
                | <g_l_arg_list> ',' <id>

<g_body> ::=
                  <g_body1> <region_stmt> <g_body2> <g_std_stmts>

<l_body> ::=
                  <g_body1> <g_body2> <std_stmts>

<g_body1> ::=
                  <g_body1> <implicit_stmt>
                | <g_body1> <spec_stmt>
                | <g_body1> 'GCONTINUE'
                | <g_body1> 'CONTINUE'
                | /* empty */

<g_body2> ::=
                  <g_body2> <spec_stmt>
                | <g_body2> <external_stmt>
                | <g_body2> 'GCONTINUE'
                | <g_body2> 'CONTINUE'
                | /* empty */
```

```
/* specification statements */
<implicit_stmt> ::=
                  'IMPLICIT' <implicit_body>

<implicit_body> ::=
                    <type> [ '(' <implicit_arg_list> ')' ]
                  | 'NONE' [ '(' <implicit_arg_list> ')' ]
                  | <implicit_body> ',' <type> [ '(' <implicit_arg_list> ')' ]
                  | <implicit_body> ',' 'NONE' [ '(' <implicit_arg_list> ')' ]

<implicit_arg_list> ::=
                  | <letter> [ '-' <letter> ]
                  | <implicit_arg_list> ',' <letter> [ '-' <letter> ]

<parameter_stmt> ::=
                  'PARAMETER' '(' <parameter_arg_list> ')'

<parameter_arg_list> ::=
                    <const_id> '=' <expr>
                  | <parameter_arg_list> ',' <const_id> '=' <expr>

<region_stmt> ::=
                  'REGION' <region_body>

<region_body> ::=
                  | <dim2> [ ',' <dim3> ]
                  | <dim3> [ ',' <dim2> ]

<spec_stmt> ::=
                    <type_stmt>
                  | <htype_stmt>
                  | <dimension_stmt>
                  | <common_stmt>
                  | <equiv_stmt>
                  | <parameter_stmt>

<type_stmt> ::=
                    <type> <type_body>

<htype_stmt> ::=
                    <htype> <htype_body>

<type_body> ::=
                    <type_entry>
                  | <type_body> ',' <type_entry>

<htype_body> ::=
                    <htype_entry>
                  | <htype_body> ',' <htype_entry>
```

```
<dimension_stmt> ::=
                  'DIMENSION' <dimension_body>

<dimension_body> ::=
                  <type_entry>
                | <dimension_body> ',' <type_entry>

<type_entry> ::=
                  <id>
                | <id> <vector_idx_bnd_list>
                | <id> <array_idx_bnd_list>
                | <proc_id>

<htype_entry> ::=
                  <id>
                | <id> <harray_idx_bnd_list>

<common_stmt> ::=
                  'COMMON' <common_body>

<common_body> ::=
                  <common_list>
                | <common_body> <common_list>

<common_list> ::=
                  '/' [ <common_id> ] '/' <common_arg_list>

<common_arg_list> ::=
                  <common_arg_entry>
                | <common_arg_list> ',' <common_arg_entry>

<common_arg_entry> ::=
                  <id>
                | <id> <vector_idx_bnd_list>
                | <id> <array_idx_bnd_list>

<equiv_stmt> ::=
                  'EQUIVALENCE' <equiv_body>

<equiv_body> ::=
                  '(' <equiv_arg_list> ')'
                | <equiv_body> ',' '(' <equiv_arg_list> ')'

<equiv_arg_list> ::=
                  <equiv_arg_entry>
                | <equiv_arg_list> ',' <equiv_arg_entry>

<equiv_arg_entry> ::=
                  <id> <array_index_list>
                | <id> <vector_index_list>
                | <id>
```

```
<external_stmt> ::=
                  'EXTERNAL' <external_body>

<external_body> ::=
                  <proc_id>
                | <external_body> ',' <proc_id>


/* Global Standard stamtments */
<g_std_stmts> ::=
                  <g_std_stmts> <g_std_stmt>
                | /* empty */

<g_std_stmt> ::=
                  <null_label> <pdo_stmt>
                | <null_label> <implicit_pdo_stmt>
                | <null_label> <pcast_stmt>
                | <null_label> <preduce_stmt>
                | <null_label> <unify_stmt>
                | <null_label> <ifall_stmt>
                | <null_label> <ifany_stmt>
                | <null_label> 'GDO' <label> <do_range>
                | <null_label> 'GGOTO' <label>
                | <null_label> <gcall_stmt>
                | <null_label> <gif_stmt>
                | <null_label> <gifthen_stmt>
                | <null_label> 'GENDDO'
                | <null_label> 'GELSE'
                | <null_label> 'GENDIF'
                | <null_label> 'GCONTINUE'
                | <null_label> 'GRETURN'

<pdo_stmt> ::=
                  'PDO' <p_range> <std_stmts> 'PEND'

<implicit_pdo_stmt> ::=
                  'PDO' <std_stmts> 'PEND'

<p_range> ::=
                  <ids> '=' <expr> ',' <expr>
                | '/' <ids> '/' '=' '/' <expr> ':' <expr> '/'
                | <p_range> ',' <ids> '=' <expr> ',' <expr>
                | <p_range> ',' '/' <ids> '/' '=' '/' <expr> ':' <expr> '/'

<pcast_stmt> ::=
                  'PCAST' <p_range> <pcast_body> 'PEND'

<pcast_body> ::=
                  <pcast_body> <pcast_assign>
                | /* empty */

<pcast_assign> ::=
```

```
                <id> <vector_index_list> '=' <id> <array_index_list>

<ifall_stmt> ::=
                'IFALL' '(' <expr> ',' <p_range> ')' 'GGOTO' <label>
              | 'IFALL' '(' <expr> ',' <p_range> ')' 'GRETURN'

<ifany_stmt> ::=
                'IFANY' '(' <expr> ',' <p_range> ')' 'GGOTO' <label>
              | 'IFANY' '(' <expr> ',' <p_range> ')' 'GRETURN'

<gif_stmt> ::=
                'GIF' '(' <expr> ')' [ 'ON' '(' <expr> ')' ] 'GGOTO' <label>
              | 'GIF' '(' <expr> ')' [ 'ON' '(' <expr> ')' ] 'GRETURN'

<gifthen_stmt> ::=
                'GIF' '(' <expr> ')' [ 'ON' '(' <expr> ')' ] 'THEN' <label>

<gcall_stmt> ::=
                'GCALL' 'PROC' <gsub_arg>

<gsub_arg> ::=
               [ '(' [ <gsub_arg_list> ] ')' ]

<gsub_arg_list> ::=
                <expr>
              | <gsub_arg_list> ',' <expr>

<preduce_stmt> ::=
                'PREDUCE' <preduce_body>

<preduce_body> ::=
                '(' <reduce_op> ',' <id> ')'
              | '(' <reduce_op> ',' <id> '(' <expr> ':' <expr> ')' ')'
              | '(' <reduce_op> ',' <id> ',' <expr> ')'
              | '(' <reduce_op> ',' <id> ',' <expr> ',' <p_range> ')'

<unify_stmt> ::=
                'UNIFY' '(' <unify_list> ')' 'ON' '(' <expr> ')'

<unify_list> ::=
                <unify_entry>
              | <unify_list> ',' <unify_entry>

<unify_entry> ::=
                <id>
              | <id> <vector_index_list>
              | <id> <vector_idx_bnd_list>


/* Local standard statements */
<std_stmts> ::=
                <std_stmts> <std_stmt>
```

```
                | /* empty */

<std_stmt> ::=
                  <null_label> <basic_stmt>
                | <null_label> <do_stmt>
                | <null_label> <cond_stmt>

<basic_stmt> ::=
                  <assign_stmt>
                | 'GOTO' <label>
                | <call_stmt>
                | 'CONTINUE'
                | 'RETURN'
                | <print_stmt>

<cond_stmt> ::=
                  <blockif_stmt>
                | <logicalif_stmt>

<do_stmt> ::=
                  'DO' [ <do_range> ] <std_stmts> 'ENDDO'

<do_range> ::=
                  <id> '=' <expr> ',' <expr> [ ',' <expr> ]

<assign_stmt> ::=
                  <variable> '=' <expr>

<logicalif_stmt> ::=
                  'IF' '(' <expr> ')' <basic_stmt>

<call_stmt> ::=
                  'CALL' <proc_id> <lcall_arg>

<lcall_arg> ::=
                  [ '(' [ <lcall_arg_list> ] ')' ]

<lcall_arg_list> ::=
                  <expr>
                | <proc_id>
                | <lcall_arg_list> ',' <expr>
                | <lcall_arg_list> ',' <proc_id>

<blockif_stmt> ::=
                  <if_then_stmt> [ <else_if_stmts> ] [ <else_stmt> ] 'ENDIF'

<if_then_stmt> ::=
                  'IF' '(' <expr> ')' 'THEN'  <std_stmts>

<else_stmt> ::=
                  'ELSE'  <std_stmts>
```

```
<else_if_stmts> ::=
                  <else_if_stmt>
                | <else_if_stmts> <else_if_stmt>

<else_if_stmt> ::=
                  'ELSEIF' '(' <expr> ')' 'THEN'  <std_stmts>

<print_stmt> ::=
                  'PRINT' '(' '*' ')'
                | 'PRINT' '(' '*' ')' ',' <print_list>
                | 'PRINT' '(' <expr> ')'
                | 'PRINT' '(' <expr> ')' ',' <print_list>

<print_list> ::=
                  <print_list_element>
                | <print_list> ',' <print_list_element>

<print_list_element> ::=
                  <expr>
                | <print_do>
                | <string>

<print_do> ::=
                  '(' <expr> ',' <id> '=' <expr> ',' <expr> [ ',' <expr> ] ')'


/* Expression */
<dim1> ::=
                  '(' <expr> ')'
<dim2> ::=
                  '(' <expr> ',' <expr> ')'
<dim3> ::=
                  '(' <expr> ',' <expr> ',' <expr> ')'

<var_dim1>::=
                  '(' '/' <expr> '/' ')'

<var_dim2> ::=
                  <dim2>
                | '(' '/' <expr> '/' ',' [ <expr> ] ')'
                | '(' [ <expr> ] ',' '/' <expr> '/' ')'
                | '(' ',' <expr> ')'
                | '(' <expr> ',' ')'

<var_dim3> ::=
                  <dim3>
                | '(' '/' <expr> ',' <expr> '/' ',' [ <expr> ] ')'
                | '(' <expr> '/' ',' [ <expr> ] ',' '/' <expr> ')'
                | '(' [ <expr> ] ',' '/' <expr> ',' <expr> '/' ')'
                | '(' ',' <expr> ',' <expr> ')'
                | '(' <expr> ',' ',' <expr> ')'
                | '(' <expr> ',' <expr> ',' ')'
```

```
<vector_index_list> ::=
                    <dim1> <expand_index_list>


<array_index_list> ::=
                    <var_dim1> <expand_index_list>
                  | <var_dim2> <expand_index_list>
                  | <var_dim3> <expand_index_list>


<expand_index_list> ::=
                    <expand_index_list> <dim1>
                  | /* empty */


<array_idx_bnd_list> ::=
                    '(' '/' <index_bound_asta> '/' ')' <expand_idx_bnd_list>
                  | '(' <index_bound_asta> ',' <index_bound_asta> ')' <expand_idx_
                    bnd_list>
                  | '(' <index_bound_asta> ',' <index_bound_asta> ',' <index_bound
                    _asta> ')' <expand_idx_bnd_list>

<vector_idx_bnd_list> ::=
                    '(' <index_bound> ')' <expand_idx_bnd_list>

<expand_idx_bnd_list> ::=
                    <expand_idx_bnd_list> '(' <index_bound> ')'
                  | /* empty */

<harray_idx_bnd_list> ::=
                    '(' <index_bound_asta> ')'
                  | '(' <index_bound> ',' <index_bound_asta> ')'
                  | '(' <index_bound> ',' <index_bound> ',' <index_bound_asta> ')'
                  | '(' <index_bound> ',' <index_bound> ',' <index_bound> ',' <ind
                    ex_bound_asta> ')'
                  | '(' <index_bound> ',' <index_bound> ',' <index_bound> ',' <ind
                    ex_bound> ',' <index_bound_asta> ')'
                  | '(' <index_bound> ',' <index_bound> ',' <index_bound> ',' <ind
                    ex_bound> ',' <index_bound> ',' <index_bound_asta> ')'
                  | '(' <index_bound> ',' <index_bound> ',' <index_bound> ',' <ind
                    ex_bound> ',' <index_bound> ',' <index_bound> ',' <index_bound
                    _asta> ')'

<index_bound> ::=
                    <expr> [ ':' <expr> ]

<index_bound_asta> ::=
                    <index_bound>
                  | '*'
```

```
<expr> ::=
                  <primary_expr>
                | <expr> '+' <expr>
                | <expr> '-' <expr>
                | <expr> '*' <expr>
                | <expr> '/' <expr>
                | <expr> '**' <expr>
                | <expr> '.EQ.' <expr>
                | <expr> '.NE.' <expr>
                | <expr> '.GT.' <expr>
                | <expr> '.GE.' <expr>
                | <expr> '.LT.' <expr>
                | <expr> '.LE.' <expr>
                | <expr> '.AND.' <expr>
                | <expr> '.OR.' <expr>
                | <expr> '.EQV.' <expr>
                | <expr> '.NEQV.' <expr>
                | '+' <expr>
                | '-' <expr>
                | '.NOT.' <expr>

<primary_expr> ::=
                  <variable>
                | <function>
                | <constant>
                | '(' <expr> ',' <expr>')'
                | '(' <expr>')'

<variable> ::=
                  <id>
                | <id> <vector_index_list>
                | <id> <array_index_list>

<function> ::=
                  <ifunc1> '(' <expr>')'
                | <ifunc2> '(' <expr> ',' <expr>')'
                | <proc_id> <lsub_arg>

<constant> ::=
                  <const_id>
                | <integer_number>
                | <real_number>
                | <complex_number>
                | <logical_number>


/* Characters, Lexical Tokens and so on */
<digit>          ::= "[0-9]"
<letter>         ::= "[A-Z]"
<id>             ::= <letter>"[A-Z0-9_]*"
<ids>            ::= <id> | <ids> ',' <id>
<const_id>       ::= <id>
```

```
<proc_id>         ::= <id>
<common_id>       ::= <id>
<integer_number>  ::= "[0-9]+"
<real_number1>    ::= "[0-9]+\." | "[0-9]+|\.[0-9]" | "\.[0-9]+"
<real_number2>    ::= <real_number1>"[ED][+-]?"<integer_number>
<real_number>     ::= <real_number1> | <real_number2>
<primary_number>  ::= <integer_number> | <real_number>
<complex_number>  ::= '(' <primary_number> ',' <primary_number> ')'
<logical_number>  ::= '.TRUE.' | '.FALSE.'
<ifunc1> ::=       'ABS'  | 'ACOS' | 'ASIN' | 'ATAN' | 'COS'  | 'COSH'
                 | 'DBLE' | 'EXP'  | 'INT'  | 'LOG'  | 'SIN'  | 'SINH'
                 | 'SQRT' | 'TAN'  | 'TANH'
<ifunc2> ::=       'ATAN2' | 'MAX'  | 'MIN'  | 'MOD'  | 'SIGN' | 'IAND'
                 | 'IOR'   | 'IXOR'
<reduce_op> ::=
                   'GSUM'   | 'GMULT'  | 'GMAX'   | 'GMIN'
                 | 'GIMAX'  | 'GIMIN'  | 'GNORM1' | 'GNORM2'
                 | 'GNORMI' | 'GIAND'  | 'GIOR'   | 'GALL'
                 | 'GANY'
<type>            ::= 'INTEGER' | 'REAL' | 'COMPLEX' | 'LOGICAL'
<htype>           ::= 'HOSTINTEGER' | 'HOSTREAL' | 'HOSTCOMPLEX' | 'HOSTLOGICAL'
<label>           ::= <digit> [ <digit> [ <digit> [ <digit> [ <digit> ]]]]
<null_label>      ::= <label> | /* empty */
```

その他の文法制約事項

コメント, 継続行等の指定はFortran90の固定形式に準拠する. 変数型REALは倍精度(8バイト)を意味し、COMPLEXも同様に倍精度(16バイト)を意味する. 名前命名規則に関してもFortran90に準拠する. 空白(white space)の扱いはFORTRANと同じであり, 空白そのものの意味はなくトークン区切子としての役割はしない.

付録 B

ADETRAN4 サンプルプログラム

本章では, ADETRAN4 サンプルプログラムとともに ADETRAN4 処理系 (ターゲット言語は MPI + Fortran 90) での出力結果を掲載する. サンプルとしてとりあげたプログラムは本文 3 章第 2 節で示した, ブロックハウスホルダー 3 重対角化アルゴリズムを ADETRAN4 で実現したものである. コンパイル出力プログラムは可読性を考慮して一部変更を加えている.

入力プログラム (ADETRAN4)

```
*
*  Tridiagonal Reduction routine
*
      gsubroutine trd(a,d_out,e_out,e2_out,n,nm,m)
*
      implicit none
      integer n,m,nm,mm
      region (nm,n)
*
      real     a(*,*),d_out(n),e_out(n),e2_out(n)
      real     d(/*/),e(/*/),e2(/*/)
      real     u(nm)(m),v(nm)(m),u_(n+1),v_(n+1),w(nm)(m)
      integer  i,j,k,L,i0,j0,k0
      integer  i_block,i_base,i0_end,ib_end
      integer  m0, m0mod4
      real     h,h1,t1,t2,t3
      real     scale,uj,vj,uj0,uj1,uj2,uj3,vj0,vj1,vj2,vj3
*
      gif ( n == 1 ) then
        pdo j=n,n
          d_out(j)=a(j,/j/)
        pend
        unify(d_out) on(1)
        e_out(n) = 0.0
        e2_out(n) = 0.0
        return
      gendif

      pdo  j = 1,n
         d(/j/)  = a(j,/j/)
         e(/j/)  = 0.0
         e2(/j/) = 0.0
      pend
      pdo  j = 1,n
         do  i = j+1, n
```

```
            a(i,/j/) = 0.0
         enddo
      pend
*
      mm     = (n-1)/m+1
      ib_end = max(1,3*(2-m))
*
      gdo  i_block = mm, ib_end, -1
*
         i_base = (i_block-1)*m
         m0     = min(m,n-i_base)
         i0_end = max(1,3*(2-i_block))
*
         pdo  j = i_base+1,i_base+m0
            j0 = j - i_base
            do  k= 1, j
               w(k)(j0) = a(k,/j/)
            enddo
         pend
*
         gdo  i0 = m0, i0_end, -1
*
            i = i0 + i_base
            L = i - 1
*
            pdo  j = i,i
               j0 = j - i_base
               scale = 0.0
               do  k = 1, L
                  scale = scale + abs(w(k)(j0))
               enddo
               if ( scale <> 0.0 ) then
                  h = 0.0
                  j0 = j - i_base
                  do  k = 1, L
                     u_(k) = w(k)(j0)/scale
                     h     = h + u_(k)**2
                  enddo
                  u_(i) = h
               endif
               u_(i+1) = scale
            pend
            unify (u_(1:i+1)) on(i)
            scale = u_(i+1)
            gif ( scale == 0.0 ) on(i) then
               pdo
                  do  j = 1, l
                     u(j)(i0) = 0.0
                     v(j)(i0) = 0.0
                  enddo
               pend
               goto 1000
            gendif
            pdo
               h  = u_(i)
               t1 = u_(L)
               t2 = - sign(sqrt(h),t1)
               u_(L) = t1 - t2
               h1    = h - t1 * t2
               do  j = 1, L
                  v_(j) = 0.0
               enddo
            pend
            pdo  j = i,i
               e(/j/)   = scale * t2
               e2(/j/)  = scale * scale * h
               j0 = j  - i_base
               w(L)(j0) = scale * u_(L)
```

```
            pend
            pdo  j = 1,L
               t1 = u_(j); t2 = v_(j)
               do  k = 1, j-1
                  v_(k) = v_(k) + a(k,/j/)*t1
                  t2    = t2 + a(k,/j/)*u_(k)
               enddo
               v_(j) = t2 + a(j,/j/)*u_(j)
            pend
            preduce(sum,v_(1:L))
            pdo
               do  k = i0+1, m0
                  t1 = 0.0; t2 = 0.0
                  do j = 1, L
                     t1 = t1 + v(j)(k)*u_(j)
                     t2 = t2 + u(j)(k)*u_(j)
                  enddo
                  do  j = 1, L
                     v_(j) = v_(j) - u(j)(k)*t1 - v(j)(k)*t2
                  enddo
               enddo
               t3 = 0.0
               do  j = 1, L
                  v_(j) = v_(j)/h1
                  t3    = t3 + v_(j)*u_(j)
               enddo
               t3 = t3/(h1+h1)
               do  j = 1, L
                  u(j)(i0) = u_(j)
                  v(j)(i0) = v_(j)-t3*u_(j)
               enddo
            pend
            gif ( i0 > 1 ) then
               pdo  j = i_base+1,i_base+i0-1
                  j0 = j - i_base
                  uj = u(j)(i0)
                  vj = v(j)(i0)
                  do  k = 1, j
                     w(k)(j0) = w(k)(j0) - (uj*v(k)(i0) + vj*u(k)(i0))
                  enddo
               pend
            gendif
*
1000     genddo
*
         pdo  j = i_base+1,i_base+m0
            j0 = j - i_base
            do  k= 1, j
               a(k,/j/) = w(k)(j0)
            enddo
         pend
*
         gif ( i_block > 1 ) then
            m0mod4=mod(m0,4)
            gdo  i0 = 1, m0mod4
               pdo  j = 1,i_base
                  uj = u(j)(i0);    vj = v(j)(i0)
                  do  k = 1, j
                     a(k,/j/) = a(k,/j/) \
                              - (uj*v(k)(i0) + vj*u(k)(i0))
                  enddo
               pend
            genddo
            gdo  i0 = 1, m0,4
               pdo  j = 1,i_base
                  uj0 = u(j)(i0+0); vj0 = v(j)(i0+0)
                  uj1 = u(j)(i0+1); vj1 = v(j)(i0+1)
                  uj2 = u(j)(i0+2); vj2 = v(j)(i0+2)
```

```
                uj3 = u(j)(i0+3); vj3 = v(j)(i0+3)
                do  k = 1, j
                   a(k,/j/) = a(k,/j/) \
                            - (uj0*v(k)(i0  ) + vj0*u(k)(i0  )) \
                            - (uj1*v(k)(i0+1) + vj1*u(k)(i0+1)) \
                            - (uj2*v(k)(i0+2) + vj2*u(k)(i0+2)) \
                            - (uj3*v(k)(i0+3) + vj3*u(k)(i0+3))
                enddo
             pend
          genddo
       gendif
*
    genddo
*
    pdo  j = 2,2
       e(/j/)   = -a(1,/j/)
       e2(/j/)  =  a(1,/j/)**2
       a(1,/j/) = 2.0 * a(1,/j/)
    pend
    pdo  j = 1,n
       d(/j/)   = a(j,/j/)
    pend
*
    pcast  i=1,n
      d_out(i)  = d(/i/)
      e_out(i)  = e(/i/)
      e2_out(i) = e2(/i/)
    pend
*
    return
    end
```

コンパイル出力プログラム (ADETRAN4)

```
      subroutine trd(p1$a,d_out,e_out,e2_out,n,nm,m)
C...Translated by AIMS 1.0 Thu Jan  9 10:05:00 1999
      implicit double precision(a-h,o-z),integer(i-n)
      double precision p1$a(*)
      double precision d_out(*),e_out(*),e2_out(*)
      double precision , pointer :: u_(:), v_(:)
      double precision , pointer :: p$d(:), p$e(:), p$e2(:)
      double precision , pointer :: u(:), v(:), w(:)
*
      logical tmp$1
      include 'mpif.h'
      common/ADE4_USEMPI/i$inod,i$nnod,i$ade4_comm_world
     1 i$status(mpi_status_size)
      data initial_flag$trd/0/
*
      call mpi_initialized(tmp$1,i$err)
      if(.not.tmp$1)then
         call mpi_init(i$err)
         i$ade4_comm_world=mpi_comm_world
      endif
      call mpi_comm_size(i$ade4_comm_world,i$nnod,i$err)
      call mpi_comm_rank(i$ade4_comm_world,i$inod,i$err)
      i$inod=i$inod+1
*
      call mpi_bcast(n,1,MPI_INTEGER,0,i$ade4_comm_world,i$err)
      call mpi_bcast(nm,1,MPI_INTEGER,0,i$ade4_comm_world,i$err)
      call mpi_bcast(m,1,MPI_INTEGER,0,i$ade4_comm_world,i$err)
*
      if(initial_flag$trd.eq.0)then
         initial_flag$trd=1
         allocate(u_(1:(n+1 )),stat=i$stat)
         if(i$stat.ne.0)initial_flag$trd=0
         allocate(v_(1:(n+1 )),stat=i$stat)
         if(i$stat.ne.0)initial_flag$trd=0
         allocate(p$e2(1:((max(nm,n)-1)/i$nnod+1)),stat=i$stat)
         if(i$stat.ne.0)initial_flag$trd=0
         allocate(p$d(1:((max(nm,n)-1)/i$nnod+1)),stat=i$stat)
         if(i$stat.ne.0)initial_flag$trd=0
         allocate(p$e(1:((max(nm,n)-1)/i$nnod+1)),stat=i$stat)
         if(i$stat.ne.0)initial_flag$trd=0
         allocate(u(1:(nm )*(m )),stat=i$stat)
         if(i$stat.ne.0)initial_flag$trd=0
         allocate(v(1:(nm )*(m )),stat=i$stat)
         if(i$stat.ne.0)initial_flag$trd=0
         allocate(w(1:(nm )*(m )),stat=i$stat)
         if(i$stat.ne.0)initial_flag$trd=0
      endif
      if(initial_flag$trd.ne.1)then
         if(i$inod.eq.1)then
            print*,"Memory allocation error."
         endif
         call mpi_abort(mpi_comm_world,1,i$err)
      endif
*
      call mpi_barrier(i$ade4_comm_world,i$err)
      call trd$body(p1$a(1),d_out(1),e_out(1),e2_out(1),n,nm,m,u_(1),
     1 v_(1),p$e2(1),p$d(1),p$e(1),u(1),v(1),w(1))
      call mpi_barrier(i$ade4_comm_world,i$err)
*
      deallocate(u_,v_,p$e2,p$d,p$e,u,v,w)
      initial_flag$trd=0
*
      return
      end
```

```
*
*
*
      subroutine trd$body(l1$a,d_out,e_out,e2_out,n,nm,m,u_,v_,l$e2,l
     1 $d,l$e,u,v,w)
C...Translated by AIMS 1.0 Thu Jan  9 10:05:00 1999
      implicit double precision(a-h,o-z),integer(i-n)
      double precision l1$a(1:nm,*)
      double precision d_out(1:n),e_out(1:n),e2_out(1:n)
      double precision u_(1:(n+1)),v_(1:(n+1))
      double precision l$d(*),l$e(*),l$e2(*)
      double precision u(1:nm,1:m),v(1:nm,1:m),w(1:nm,1:m)
*
      include 'mpif.h'
      common/ADE4_USEMPI/i$inod,i$nnod,i$ade4_comm_world
     1 ,i$status(mpi_status_size)
*
      if(n.EQ.1)then
*
         i$2=n
         if(i$2.ge.1.and.i$2.le.n)then
            if(mod(i$2-1,i$nnod)+1.eq.i$inod)then
               i$1=(i$2-1)/i$nnod+1
               d_out(n)=l1$a(n,i$1)
            endif
         endif
*
         i$2=1
         i$3=mod(i$2-1+i$nnod,i$nnod)
         i$4=(n )
         call mpi_bcast(d_out(1),i$4,MPI_DOUBLE_PRECISION,i$3,i$ade4_
     1    comm_world,i$err)
*
         e_out(n)=0.0D+00
         e2_out(n)=0.0D+00
*
         goto 99999
      endif
*
      i$2=1
      i$3=n
      i$1=n
      call loop$cyclic(i$2,i$3,i$4,i$1)
      do i$1=i$2,i$3
         j=(i$1-1)*i$nnod+i$inod
         l$d(i$1)=l1$a(j,i$1)
         l$e(i$1)=0.0D+00
         l$e2(i$1)=0.0D+00
      enddo
*
      i$2=1
      i$3=n
      i$1=n
      call loop$cyclic(i$2,i$3,i$4,i$1)
      do i$1=i$2,i$3
         j=(i$1-1)*i$nnod+i$inod
         do i=j+1,n,1
            l1$a(i,i$1)=0.0D+00
         enddo! i
      enddo
*
      mm=(n-1)/m+1
      ib_end=Max(1,3*(2-m))
*
      do i_block=mm,ib_end,-1
*
         i_base=(i_block-1)*m
         m0=Min(m,n-i_base)
```

```
      i0_end=Max(1,3*(2-i_block))
*
      i$2=i_base+1
      i$3=i_base+m0
      i$1=n
      call loop$cyclic(i$2,i$3,i$4,i$1)
      do i$1=i$2,i$3
         j=(i$1-1)*i$nnod+i$inod
         j0=j-i_base
         do k=1,j,1
            w(k,j0)=l1$a(k,i$1)
         enddo! k
      enddo
*
      do i0=m0,i0_end,-1
         i=i0+i_base
         l=i-1
*
         i$2=i
         if(i$2.ge.1.and.i$2.le.max(nm,n))then
            if(mod(i$2-1,i$nnod)+1.eq.i$inod)then
               i$1=(i$2-1)/i$nnod+1
               j=i
               j0=j-i_base
               scale=0.0D+00
               do k=1,l,1
                  scale=scale+Abs(w(k,j0))
               enddo! k
               if(scale.NE.0.0D+00)then
                  h=0.0D+00
                  j0=j-i_base
                  do k=1,l,1
                     u_(k)=w(k,j0)/scale
                     h=h+u_(k)**2
                  enddo! k
                  u_(i)=h
               endif
               u_(i+1)=scale
            endif
         endif
*
         i$2=i
         i$3=mod(i$2-1+i$nnod,i$nnod)
         i$5=1
         i$6=i+1
         i$4=i$6-i$5+1
         call mpi_bcast(u_(1),i$4,MPI_DOUBLE_PRECISION,i$3,i$ade4_
     1   comm_world,i$err)
*
         scale=u_(i+1)
         if(scale.EQ.0.0D+00)then
*
            do j=1,l,1
               u(j,i0)=0.0D+00
               v(j,i0)=0.0D+00
            enddo! j
*
            goto 1000
         endif
*
         h=u_(i)
         t1=u_(l)
         t2=-Sign(Sqrt(h),t1)
         u_(l)=t1-t2
         h1=h-t1*t2
         do j=1,l,1
            v_(j)=0.0D+00
         enddo! j
```

```
*
            i$2=i
            if(i$2.ge.1.and.i$2.le.n)then
               if(mod(i$2-1,i$nnod)+1.eq.i$inod)then
                  i$1=(i$2-1)/i$nnod+1
                  j=i
                  l$e(i$1)=scale*t2
                  l$e2(i$1)=scale*scale*h
                  j0=j-i_base
                  w(l,j0)=scale*u_(l)
               endif
            endif
*
            i$2=1
            i$3=l
            i$1=n
            call loop$cyclic(i$2,i$3,i$4,i$1)
            do i$1=i$2,i$3
               j=(i$1-1)*i$nnod+i$inod
               t1=u_(j); t2=v_(j)
               do k=1,j-1,1
                  v_(k)=v_(k)+l1$a(k,i$1)*t1
                  t2=t2+l1$a(k,i$1)*u_(k)
               enddo! k
               v_(j)=t2+l1$a(j,i$1)*u_(j)
            enddo
*
            call preduce$sum_real(v_(1),l)
*
            do k=i0+1,m0,1
               t1=0.0D+00; t2=0.0D+00
               do j=1,l,1
                  t1=t1+v(j,k)*u_(j)
                  t2=t2+u(j,k)*u_(j)
               enddo! j
               do j=1,l,1
                  v_(j)=v_(j)-u(j,k)*t1-v(j,k)*t2
               enddo! j
            enddo! k
            t3=0.0D+00
            do j=1,l,1
               v_(j)=v_(j)/h1
               t3=t3+v_(j)*u_(j)
            enddo! j
            t3=t3/(h1+h1)
            do j=1,l,1
               u(j,i0)=u_(j)
               v(j,i0)=v_(j)-t3*u_(j)
            enddo! j
*
            if(i0.GT.1)then
*
               i$2=i_base+1
               i$3=i_base+i0-1
               i$1=2**30
               call loop$cyclic(i$2,i$3,i$4,i$1)
               do i$1=i$2,i$3
                  j=(i$1-1)*i$nnod+i$inod
                  j0=j-i_base
                  uj=u(j,i0); vj=v(j,i0)
                  do k=1,j,1
                     w(k,j0)=w(k,j0)-(uj*v(k,i0)+vj*u(k,i0))
                  enddo! k
               enddo
*
            endif
1000        continue
         enddo
```

```
*
          i$2=i_base+1
          i$3=i_base+m0
          i$1=n
          call loop$cyclic(i$2,i$3,i$4,i$1)
          do i$1=i$2,i$3
             j=(i$1-1)*i$nnod+i$inod
             j0=j-i_base
             do k=1,j,1
                l1$a(k,i$1)=w(k,j0)
             enddo! k
          enddo
*
          if(i_block.GT.1)then
*
             m0mod4=Mod(m0,4)
*
             do i0=1,m0mod4,1
*
                i$2=1
                i$3=i_base
                i$1=n
                call loop$cyclic(i$2,i$3,i$4,i$1)
                do i$1=i$2,i$3
                   j=(i$1-1)*i$nnod+i$inod
                   uj=u(j,i0); vj=v(j,i0)
                   do k=1,j,1
                      l1$a(k,i$1)=l1$a(k,i$1)-(uj*v(k,i0)+vj*u(k,i0))
                   enddo! k
                enddo
*
             enddo
             do i0=1,m0,4

                i$2=1
                i$3=i_base
                i$1=n
                call loop$cyclic(i$2,i$3,i$4,i$1)
                do i$1=i$2,i$3
                   j=(i$1-1)*i$nnod+i$inod
                   uj0=u(j,i0 );  vj0=v(j,i0 )
                   uj1=u(j,i0+1); vj1=v(j,i0+1)
                   uj2=u(j,i0+2); vj2=v(j,i0+2)
                   uj3=u(j,i0+3); vj3=v(j,i0+3)
                   do k=1,j,1
                      l1$a(k,i$1)=l1$a(k,i$1)
     1                   -(uj0*v(k,i0  )+vj0*u(k,i0  ))
     2                   -(uj1*v(k,i0+1)+vj1*u(k,i0+1))
     3                   -(uj2*v(k,i0+2)+vj2*u(k,i0+2))
     4                   -(uj3*v(k,i0+3)+vj3*u(k,i0+3))
                   enddo! k
                enddo
*
             enddo
          endif
       enddo
*
       i$2=2
       if(i$2.ge.1.and.i$2.le.n)then
          if(mod(i$2-1,i$nnod)+1.eq.i$inod)then
             i$1=(i$2-1)/i$nnod+1
             l$e(i$1)=-l1$a(1,i$1)
             l$e2(i$1)=l1$a(1,i$1)**2
             l1$a(1,i$1)=2.0D+00*l1$a(1,i$1)
          endif
       endif
*
       i$2=1
```

```
      i$3=n
      i$1=n
      call loop$cyclic(i$2,i$3,i$4,i$1)
      do i$1=i$2,i$3
         j=(i$1-1)*i$nnod+i$inod
         l$d(i$1)=l1$a(j,i$1)
      enddo
*
      call pcast$cyclic$real(d_out(1),l$d(1),1,1,n,0,1)
      call pcast$cyclic$real(e_out(1),l$e(1),1,1,n,0,1)
      call pcast$cyclic$real(e2_out(1),l$e2(1),1,1,n,0,1)
*
99999 continue
      return
      end
```